

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/45173506>

# Diseño de circuitos digitales con VHDL

Article · January 2010

Source: OAI

---

CITATIONS

0

---

READS

2,216

2 authors:



[Felipe Machado](#)

King Juan Carlos University

26 PUBLICATIONS 144 CITATIONS

[SEE PROFILE](#)



[S. Borromeo](#)

King Juan Carlos University

35 PUBLICATIONS 208 CITATIONS

[SEE PROFILE](#)

# Diseño de circuitos digitales con VHDL



**Felipe Machado Sánchez  
Susana Borromeo López**

Departamento de Tecnología Electrónica  
Universidad Rey Juan Carlos

## Diseño de circuitos digitales con VHDL

Felipe Machado, Susana Borromeo

Versión 1.01 creada el 20 de julio de 2010



Esta versión digital de *Diseño de circuitos digitales con VHDL* ha sido creada y licenciada por Felipe Machado Sánchez y Susana Borromeo López con una licencia de Creative Commons. Esta licencia permite los usos no comerciales de esta obra en tanto en cuanto se atribuya autoría original. Esta licencia no permite alterar, transformar o generar una obra derivada a partir de esta obra

Con esta licencia eres libre de copiar, distribuir y comunicar públicamente esta obra bajo las condiciones siguientes:

**Reconocimiento:** debes reconocer y citar a los autores de esta obra

**No comercial:** no puedes utilizar esta obra para fines comerciales

**Sin obras derivadas:** No se puede alterar, transformar o generar una obra derivada a partir de esta obra

Para más información sobre la licencia, visita el siguiente enlace:

<http://creativecommons.org/licenses/by-nc-nd/3.0/>

Esta obra está disponible en el archivo abierto de la Universidad Rey Juan Carlos:

<http://ecienzia.urjc.es/dspace/handle/10115/4045>

<http://hdl.handle.net/10115/4045>

Para más información sobre los autores:

<http://gtebim.es/~fmachado>

<http://gtebim.es/>

**ISBN: 978-84-693-4652-5**

# Diseño de circuitos digitales con VHDL

Versión 1.01 creada el 20 de julio de 2010

Felipe Machado Sánchez

Susana Borromeo López

Departamento de Tecnología Electrónica

Universidad Rey Juan Carlos

Móstoles, Madrid, España

<http://gtebim.es/>

<http://gtebim.es/~fmachado>



*A mi hermano Ale*  
FMS



## **Agradecimientos**

*Queremos agradecer al Departamento de Tecnología Electrónica de la Universidad Rey Juan Carlos por fomentar la docencia de calidad y su apoyo constante en nuestras labores docentes.*

*También queremos agradecer a los alumnos de Ingeniería de Telecomunicación por su interés generalizado en aprender y sus comentarios sobre nuestra docencia y prácticas*

*Los Autores*



# Índice

Índice .....	1
Lista de acrónimos .....	5
Índice de figuras .....	6
Índice de código VHDL .....	9
Índice de tablas .....	11
1. Introducción .....	13
2. Encender un LED .....	15
2.1. Tarjetas Pegasus y Basys .....	15
2.1.1. La tarjeta <i>Pegasus</i> .....	15
2.1.2. La tarjeta <i>Basys</i> .....	16
2.2. Cómo encender un LED .....	17
2.3. Diseño del circuito .....	18
2.4. Síntesis e implementación del circuito .....	24
2.5. Programación de la FPGA .....	26
2.5.1. Programación de la tarjeta <i>Pegasus</i> .....	26
2.5.2. Programación de la tarjeta <i>Basys</i> .....	29
2.6. Cambiar el tipo de FPGA de un proyecto .....	31
2.7. Trabajar desde varios ordenadores con un mismo proyecto .....	32
2.8. Conclusión .....	33
3. Sentencias concurrentes .....	34
3.1. Diseño de un multiplexor .....	35
3.1.1. Diseño usando puertas lógicas .....	35
3.1.2. Diseño usando varias sentencias concurrentes .....	36
3.1.3. Uso de sentencias condicionales .....	37
3.1.4. Uso de procesos .....	37
3.2. Diseño de un multiplexor de 4 alternativas .....	38
3.3. Diseño de un multiplexor de 4 bits de dato y dos alternativas .....	39
3.4. Conclusiones .....	40
4. Codificadores, decodificadores, convertidores de código .....	41
4.1. Convertidor de binario a 7 segmentos .....	41
4.2. Decodificador de 2 a 4 .....	43
4.3. Codificador de 8 a 3 .....	45
4.4. Conclusiones .....	48
5. Elementos de memoria .....	50
5.1. Biestable J-K .....	50
5.2. Descripción de biestables en VHDL para síntesis .....	51
5.3. Encendido y apagado de LED con un pulsador .....	54
5.3.1. Primera solución .....	55
5.3.2. Detector de flanco .....	55
5.4. Conclusiones .....	59
6. Contadores .....	61
6.1. Segundero .....	61
6.2. Contador de 10 segundos .....	63
6.3. Cronómetro .....	65
6.3.1. Mostrar los dígitos. Solución manual .....	67
6.3.2. Mostrar los dígitos. Solución automática .....	68
6.3.3. Mejoras del circuito .....	70
6.3.4. Optimización .....	70
6.4. Contador manual .....	71
6.5. Conclusiones .....	72
7. Registros de desplazamiento .....	73
7.1. Registro de desplazamiento con carga paralelo .....	73
7.2. Rotación a la derecha y a la izquierda .....	75
7.3. Rotación automática .....	76
7.3.1. Variantes del circuito .....	76
7.4. Conclusiones .....	76
8. Simulación .....	77
8.1. Ejemplo sencillo .....	78
8.2. Ampliación .....	86
8.3. Conclusiones .....	86
9. Máquinas de estados finitos .....	87
9.1. Máquina de estados para encender y apagar un LED con pulsador .....	87

9.1.1. Proceso combinacional que obtiene el estado siguiente .....	89
9.1.2. Proceso secuencial .....	89
9.1.3. Proceso combinacional que proporciona las salidas .....	89
9.2. Detector de flanco con máquinas de estados .....	90
9.2.1. Variante .....	91
9.3. Desplazamiento alternativo de los LED .....	91
9.4. Conclusiones .....	93
10. Clave electrónica .....	95
10.1. Variantes .....	96
11. Circuito antirrebotes .....	97
12. Máquina expendedora .....	99
12.1. Versión sencilla .....	99
12.2. Versión decimal .....	100
13. Control con PWM .....	101
13.1. Funcionamiento del PWM .....	101
13.2. Control de la intensidad de un LED .....	101
13.3. Ampliación a control de motores .....	102
Circuitos digitales y analógicos.....	103
14. Control de motor paso a paso .....	105
14.1. Motores paso a paso .....	105
14.1.1. Identificación de terminales .....	105
14.1.2. Secuencia del motor paso a paso .....	107
14.2. Generación de la secuencia de control con la FPGA .....	108
14.3. Circuito de potencia para gobernar el motor .....	109
14.4. Conclusiones .....	111
15. Piano electrónico .....	113
15.1. Ampliaciones .....	114
Problemas teóricos.....	115
16. Reproductor MP3 .....	117
16.1. Enunciado .....	117
16.2. Solución .....	118
16.2.1. Bloques internos del circuito .....	118
16.2.2. Diagrama de transición de estados .....	118
16.2.3. Tabla de estados siguientes y salidas .....	121
16.2.4. Tabla de excitación de los biestables para biestables J-K .....	122
16.2.5. Ecuación simplificada para la salida Play .....	123
16.2.6. Modelo VHDL .....	123
16.3. Solución alternativa .....	125
17. Visualización del reproductor MP3 .....	129
17.1. Enunciado .....	129
17.2. Solución .....	130
17.2.1. Esquema interno del circuito .....	130
17.2.2. Modelo en VHDL .....	130
18. Robot rastreador .....	133
18.1. Enunciado .....	133
18.2. Solución .....	135
18.2.1. Entradas, salidas y estados. Diagrama de estados .....	135
18.2.2. Tabla de estados siguientes y salidas .....	137
18.2.3. Tabla de excitación de biestables para biestables J-K .....	138
18.2.4. Ecuaciones simplificadas .....	139
19. Teclado de teléfono móvil .....	141
19.1. Enunciado .....	141
19.2. Solución .....	143
19.2.1. Bloques internos del circuito .....	143
19.2.2. Diagrama de estados .....	143
20. Cálculo de temporización de un circuito 1 .....	149
20.1. Enunciado .....	149
20.2. Solución .....	149
20.2.1. Metaestabilidad .....	149
20.2.2. Frecuencia máxima .....	149
21. Análisis de un circuito 1 .....	151
21.1. Enunciado .....	151
21.2. Solución .....	151
21.2.1. Tabla de excitación del autómata .....	151
21.2.2. Diagrama de transición de estados .....	152
22. Cálculo de temporización de un circuito 2 .....	153
22.1. Enunciado .....	153
22.2. Solución .....	153

---

23. 24. Análisis de un circuito 2 .....	155
24.1. Enunciado .....	155
24.2. Solución.....	155
24.2.1. Tabla de excitación del autómata .....	155
24.2.2. Diagrama de transición de estados .....	156
Referencias .....	157



## Listado de acrónimos

ALU	<i>Arithmetic Logic Unit</i> Unidad Aritmético Lógica
CAD	<i>Computer Aided Design</i> Diseño asistido por ordenador
CPLD	<i>Complex Programmable Logic Device</i> Dispositivo de lógica programable complejo
DTE	<i>Departamento de Tecnología Electrónica</i>
ED1	<i>Electrónica Digital I</i> Asignatura de la titulación de Ingeniería de Telecomunicación de la URJC
ED2	<i>Electrónica Digital II</i> Asignatura de la titulación de Ingeniería de Telecomunicación de la URJC
ETSIT	<i>Escuela Técnica Superior de Ingeniería de Telecomunicación</i> Escuela de la Universidad Rey Juan Carlos
FPGA	<i>Field Programmable Gate Array</i> Dispositivo de lógica programable, de mayores prestaciones que los CPLD
FSM	<i>Finite State Machine</i> Máquina de estados finitos
IEEE	<i>Institute of Electrical and Electronics Engineers</i> Instituto de Ingenieros Eléctricos y Electrónicos
LED	<i>Light-Emitting Diode</i> Diodo emisor de luz
LSB	<i>Least Significant Bit</i> Bit menos significativo
MSB	<i>Most Significant Bit</i> Bit más significativo
PWM	<i>Pulse Width Modulation</i> Modulación por ancho de pulso
URJC	<i>Universidad Rey Juan Carlos</i> Universidad pública de Madrid, España
UUT	<i>Unit Under Test</i> Unidad bajo prueba
VHDL	<i>VHSIC Hardware Description Language</i> Un tipo de lenguaje de descripción de hardware
VHSIC	<i>Very High Speed Integrated Circuit</i> Circuito integrado de muy alta velocidad

## Índice de figuras

Figura 2.1: Placa Pegasus de Digilent .....	16
Figura 2.2: Placa Basys de Digilent .....	17
Figura 2.3: Conexión de los LED de la placa Pegasus (izquierda) y la Basys (derecha) .....	17
Figura 2.4: Ventana para la creación de nuevo proyecto .....	19
Figura 2.5: Interpretación del texto del encapsulado de la FPGA de la Pegasus .....	19
Figura 2.6: Interpretación del texto del encapsulado de la FPGA de la Basys.....	19
Figura 2.7: Ventana para la selección del dispositivo del nuevo proyecto.....	20
Figura 2.8: Selección del tipo de la nueva fuente que vamos a crear .....	20
Figura 2.9: Definición de los puertos .....	21
Figura 2.10: Apariencia de la herramienta al añadir la nueva fuente led0.vhd.....	21
Figura 2.11: Apariencia de la herramienta para empezar a realizar la descripción VHDL.....	22
Figura 2.12: "Caja negra" que define la entidad de nuestro circuito.....	23
Figura 2.13: Selección para la asignación de pines .....	24
Figura 2.14: Herramienta PACE para la asignación de pines.....	25
Figura 2.15: Aspecto de la subventana de procesos una vez que se ha sintetizado e implementado el diseño correctamente. En la ventana de la derecha, en el proceso de síntesis ha habido advertencias (warnings) que no siempre son importantes, y por eso tiene un símbolo triangular de color amarillo .....	26
Figura 2.16: Jumpers y su conexión .....	26
Figura 2.17: iMPACT: componentes detectados y asignación de un fichero de configuración.....	27
Figura 2.18: Orden para programar la FPGA.....	28
Figura 2.19: Programación realizada con éxito .....	29
Figura 2.20: Pantalla inicial del Adept, sin dispositivos conectados .....	30
Figura 2.21: Pantalla del Adept que ha detectado algún dispositivos conectado .....	30
Figura 2.22: Pantalla del Adept que ha detectado la cadena JTAG de dispositivos (FPGA y PROM) .....	31
Figura 2.23: Procedimiento para cambiar las características de la FPGA.....	32
Figura 3.1: Esquema eléctrico de las conexiones de los interruptores y pulsadores en la placa Pegasus .....	34
Figura 3.2: Esquema eléctrico de las conexiones de los interruptores y pulsadores en la placa Basys .....	34
Figura 3.3: Lista de palabras reservadas del VHDL.....	35
Figura 3.4: Esquema del multiplexor (izquierda) y diseño en puertas (derecha) .....	35
Figura 3.5: Esquema en puertas del multiplexor con señales intermedias.....	36
Figura 3.6: Multiplexor de 4 alternativas .....	38
Figura 3.7: Definición de los puertos de entrada y su ancho.....	38
Figura 3.8: Multiplexor de 4 bits de dato .....	39
Figura 4.1: Multiplexor y salida a 7 segmentos.....	41
Figura 4.2: Los 16 números representables con 4 bits .....	42
Figura 4.3: Valores del vector de los segmentos (SEG) para los números 0, 1 y 2.....	42
Figura 4.4: Sentencias concurrentes de la arquitectura.....	43
Figura 4.5: Puertos del decodificador y tabla de verdad.....	44
Figura 4.6: Esquema del circuito del decodificador completo.....	44
Figura 4.7: Esquema y tabla de verdad del codificador de 8 a 3.....	46
Figura 4.8: Esquema de bloques del codificador .....	46
Figura 4.9: Sentencias VHDL del esquema de la figura 4.8 .....	47
Figura 5.1: Tabla de verdad del biestable J-K (izquierda) y circuito que queremos implementar (derecha) .....	50
Figura 5.2: Editor de restricciones del ISE (Xilinx Constraints Editor) .....	51
Figura 5.3: Tabla de verdad del biestable T y su esquema.....	55
Figura 5.4: Cronograma resultante de pulsar el botón de encendido/apagado .....	56
Figura 5.5: Circuito detector de flanco sin registrar la entrada (Mealy: no recomendado para este caso) .....	56
Figura 5.6: Cronograma del detector de flanco realizado por Mealy (no recomendado para este caso) .....	57
Figura 5.7: Circuito detector de flanco registrando la entrada (Moore: recomendado para este caso) .....	57
Figura 5.8: Cronograma del detector de flanco realizado por Moore (recomendado para este caso) .....	57
Figura 5.9: Circuito que enciende y apaga un LED con el mismo pulsador .....	58
Figura 5.10: Código VHDL del circuito completo .....	58
Figura 6.1: Cronograma de la señal que queremos obtener.....	61
Figura 6.2: Esquema del circuito que queremos realizar.....	61
Figura 6.3: Esquema del contador de 10 segundos .....	63
Figura 6.4: Entradas y salidas del circuito .....	65
Figura 6.5: Cronograma de la señal de una décima de segundo que queremos obtener.....	65
Figura 6.6: Esquema del cronómetro .....	66

Figura 6.7: Esquema del proceso contador de décimas .....	66
Figura 6.8: Codificador para controlar los ánodos de los displays.....	67
Figura 6.9: Multiplexor que selecciona la cifra BCD según la configuración de os interruptores.....	67
Figura 6.10:Esquema completo del cronómetro con visualización manual .....	68
Figura 6.11 : Cronograma de la señal de una milésima de segundo que queremos obtener .....	68
Figura 6.12:Multiplexado en el tiempo para mostrar los displays .....	69
Figura 6.13:Esquema completo del cronómetro con visualización automática.....	70
Figura 6.14:Otra alternativa al esquema de la figura 6.9 .....	71
Figura 6.15: Diagrama de estados del contador ascendente/descendente que vamos a realizar .....	71
Figura 6.16: Esquema del contador que implementaremos.....	72
Figura 7.1: Esquema del registro de desplazamiento a la izquierda con carga paralelo .....	73
Figura 7.2: Desplazamiento a la izquierda.....	74
Figura 7.3: Rotación a la izquierda.....	75
Figura 7.4: Esquema del circuito de rotación automática .....	76
Figura 8.1: Esquema general de un banco de pruebas.....	78
Figura 8.2: Puertos de entrada y salida del contador que vamos a probar .....	78
Figura 8.3: Selección de nueva fuente de tipo VHDL Test Bench.....	79
Figura 8.4: Código creado automáticamente por el ISE para el banco de pruebas.....	80
Figura 8.5: Distintas partes de la referencia ("instanciación") a un componente.....	81
Figura 8.6: Periodo de la señal de reloj.....	81
Figura 8.7: Selección del entorno para simulación .....	82
Figura 8.8: Entorno para simulación.....	83
Figura 8.9: Cambio de las propiedades del proyecto.....	84
Figura 8.10: Selección del ISE Simulator .....	84
Figura 8.11: Arranque de la simulación .....	85
Figura 8.12: Resultado de la simulación.....	85
Figura 8.13: Visualización de cuenta como decimal sin signo.....	86
Figura 9.1: Esquema general de los procesos de una máquina de estados .....	87
Figura 9.2: Bloques del circuito para encender y apagar un LED.....	88
Figura 9.3: Diagrama de transición de estados para encender un LED con un pulsador.....	88
Figura 9.4: Diagrama de estados del detector de flancos (Moore).....	90
Figura 9.5: Cronograma de cuando llega la secuencia 0101.....	91
Figura 9.6: Diagrama de estados modificado .....	91
Figura 9.7: Esquema del circuito del movimiento alternativo de los LED .....	92
Figura 9.8: Diagrama de transición de estados .....	93
Figura 10.1: Esquema del circuito de la clave electrónica .....	96
Figura 11.1: Esquema de los pulsadores en la placa Pegasus y en la placa Basys .....	97
Figura 11.2: Cronograma de la salida del detector de flancos (PULSO_BTN) cuando hay rebotes en la entrada (BTN) .....	97
Figura 11.3: Cronograma de la salida del filtro antirrebotes (FILTRO_BTN) cuando hay rebotes en la entrada (BTN) .....	98
Figura 11.4: Esquema del circuito que filtra los rebotes de la entrada.....	98
Figura 11.5: Diagrama de estados del circuito que filtra los pulsos .....	98
Figura 12.1: Esquema de puertos de entrada y salida del control de la máquina expendedora .....	99
Figura 12.2: Esquema interno del circuito de control de la máquina expendedora simple .....	100
Figura 12.3: Indicaciones de los displays según el estado.....	100
Figura 13.1: Señal PWM que se le ha cambiado el ciclo de trabajo.....	101
Figura 13.2: Control de velocidad con el PWM .....	101
Figura 13.3: Los 8 ciclos de trabajo posibles con un PWM de 3 bits de resolución.....	102
Figura 13.4: Circuito de potencia para controlar el motor desde la FPGA.....	102
Figura 14.1: Conexión interna de un motor unipolar .....	106
Figura 14.2: El terminal A a tierra .....	106
Figura 14.3: Resultado de poner a tierra cada uno de los otros tres terminales, cuando el motor pierde par indica que es el correspondiente .....	106
Figura 14.4: Identificación de los terminales en nuestro motor PM55L-048 de Minebea .....	107
Figura 14.5: Secuencia de terminales que se ponen a tierra para obtener el giro del motor .....	107
Figura 14.6: Secuencia de terminales que se ponen a tierra para obtener el giro del motor en ambos sentidos .....	107
Figura 14.7: Bloque de control de motor de la FPGA .....	108
Figura 14.8: Esquema de la máquina de estados para el bloque de control de motor de la FPGA .....	109
Figura 14.9: Posible esquema de la electrónica de potencia para manejar el motor .....	110
Figura 14.10: Esquema del circuito con optoacopladores .....	110
Figura 14.11: Esquema del circuito usando el circuito integrado SN754410 .....	111
Figura 15.1: Esquema del circuito de piano electrónico básico .....	113
Figura 15.2: Esquema general del circuito de amplificación .....	114
Figura 16.1: Bloques internos del circuito .....	118
Figura 16.2: Orden de entradas y salidas en la representación de la máquina de estados.....	119

Figura 16.3: Estado inicial.....	119
Figura 16.4: Transición a un estado de espera .....	119
Figura 16.5: Transición a estado de reproducción o avance .....	120
Figura 16.6: Estado de reproducción y avance.....	120
Figura 16.7: Transición desde el estado de avance al estado de reproducción.....	120
Figura 16.8: Diagrama final de la máquina de estados.....	121
Figura 16.9: Diagrama de Karnaugh y ecuación de la salida Play .....	123
Figura 16.10: Diagrama de bloques de la solución alternativa.....	125
Figura 16.11: Diagrama de bloques más detallado de la solución alternativa.....	126
Figura 16.12: Orden de entradas y salidas en la representación de la primera máquina de estados .....	126
Figura 16.13: Diagrama de la primera máquina de estados de la solución alternativa.....	126
Figura 16.14: Orden de entradas y salidas en la representación de la segunda máquina de estados.....	127
Figura 16.15: Diagrama de la segunda máquina de estados de la solución alternativa.....	127
Figura 17.1: Texto que deben de mostrar los displays de siete segmentos .....	129
Figura 17.2: Segmentos del display.....	129
Figura 17.3: Esquema del circuito .....	129
Figura 17.4: Esquema del circuito de visualización .....	130
Figura 18.1: Coche visto desde arriba.....	133
Figura 18.2: Esquema de entradas y salidas del sistema de control .....	133
Figura 18.3: Distintas posiciones del coche respecto a la línea.....	134
Figura 18.4: Posición de los receptores respecto a la línea y giro .....	135
Figura 18.5: Orden de entradas y salidas en el diagrama de estados.....	136
Figura 18.6: Estado inicial y sus transiciones .....	136
Figura 18.7: Estados con un detector sobre la línea y otro fuera.....	136
Figura 18.8: Diagrama de estados final sin reducir .....	137
Figura 18.9: Diagrama de estados final reducido .....	137
Figura 18.10: Mapas de Karnaugh y ecuaciones de las entradas de los biestables .....	139
Figura 19.1: Esquema de entradas y salidas del sistema de control .....	141
Figura 19.2: Bloques internos del circuito .....	143
Figura 19.3: Orden de entradas y salidas en el diagrama de estados.....	143
Figura 19.4: Orden de entradas y salidas en el diagrama de estados. Salidas con iniciales para simplificar.....	143
Figura 19.5: Estado inicial.....	143
Figura 19.6: Estado de espera.....	144
Figura 19.7: Estado de espera a que suelte la tecla .....	144
Figura 19.8: Se fija el dos en el caso improbable que suelte la tecla a la vez que termina la temporización .....	145
Figura 19.9: Se suelta la tecla antes de que pase un segundo: nueva espera .....	145
Figura 19.10: Nueva espera a que se suelte B .....	146
Figura 19.11: Se ha soltado P antes de que pase un segundo .....	146
Figura 19.12: En la espera a ver si se vuelve a pulsar P mostrando la B.....	147
Figura 19.13: Diagrama de estados final .....	148
Figura 20.1: Circuito para analizar.....	149
Figura 21.1: Diagrama de transición de estados .....	152
Figura 22.1: Circuito para analizar.....	153
Figura 24.1: Circuito para analizar.....	155
Figura 24.2: Diagrama de transición de estados .....	156

## Índice de código VHDL

Código 2-1: Arquitectura que enciende un LED y apaga otro .....	23
Código 3-1: Sentencia concurrente que define el multiplexor en puertas .....	36
Código 3-2: Sentencias concurrentes que definen el multiplexor en puertas (equivalente al código 3-1) .....	36
Código 3-3: Otra versión de las sentencias concurrentes que definen el multiplexor en puertas (equivalente a los códigos 3-1 y 3-2) .....	37
Código 3-4: Sentencia concurrente condicionada que define el multiplexor .....	37
Código 3-5: Sentencia concurrente condicionada que define el multiplexor .....	37
Código 3-6: Entidad del multiplexor de 4 alternativas .....	38
Código 3-7: Sentencia concurrente condicionada que define el multiplexor de cuatro alternativas .....	39
Código 3-8: Proceso con sentencia IF .....	39
Código 3-9: Proceso con sentencia CASE .....	39
Código 3-10: Entidad del multiplexor de 4 alternativas .....	39
Código 4-1: Decodificador con sentencia concurrente .....	45
Código 4-2: Decodificador con proceso .....	45
Código 4-3: Una forma de asignar el mismo valor a todos los bits de un vector .....	45
Código 4-4: Habilitación con señal auxiliar .....	47
Código 4-5: Habilitación dentro de la sentencia .....	48
Código 4-6: Otra alternativa al circuito de la figura 4.8, todo en un mismo proceso .....	48
Código 5-1: Proceso que implementa un biestable J-K .....	50
Código 5-2: Otra alternativa para el biestable J-K .....	50
Código 5-3: Sentencia de reloj CORRECTA .....	52
Código 5-4: sentencia de reloj INCORRECTA .....	52
Código 5-5: Inicialización síncrona .....	52
Código 5-6: Asignación antes de la inicialización. INCORRECTA .....	52
Código 5-7: Asignación fuera del reloj. INCORRECTA .....	52
Código 5-8: Asignación después de la sentencia de reloj. INCORRECTA .....	53
Código 5-9: Lista de sensibilidad con el reloj y las señales asíncronas .....	53
Código 5-10: Proceso equivalente al del código 5-11 .....	53
Código 5-11: Proceso equivalente al del código 5-10 .....	53
Código 5-12: Proceso combinacional (no genera latch). Correcto .....	54
Código 5-13: Proceso que genera latch. Posiblemente no lo quieras generar .....	54
Código 5-14: Proceso combinacional (no genera latch) .....	54
Código 5-15: Proceso que es posible que genere latch. No recomendado .....	54
Código 5-16: Proceso combinacional equivalente al código 5-12 (no genera latch) .....	54
Código 5-17: Proceso combinacional equivalente al código 5-14 (no genera latch) .....	54
Código 5-18: Proceso que implementa un biestable T .....	55
Código 5-19: Alternativa que describe el biestable T .....	55
Código 5-20: Proceso que implementa los 2 biestables D .....	59
Código 5-21: Proceso equivalente al 5-20 .....	59
Código 6-1: Arquitectura del contador .....	62
Código 6-2: Proceso equivalente a los dos procesos del código 6-1 .....	63
Código 6-3: Declaración de señal unsigned .....	64
Código 6-4: Bibliotecas por defecto que pone el ISE .....	64
Código 6-5: Bibliotecas recomendadas .....	64
Código 6-6: Proceso que cuenta 10 segundos .....	65
Código 6-7: Proceso contador de décimas .....	66
Código 6-8: Código del contador ascendente/descendente que no se desborda .....	72
Código 7-1: Código del proceso de registro de desplazamiento .....	74
Código 7-2: Código equivalente al código 7-1 .....	74
Código 7-3: Código equivalente al código 7-1 .....	74
Código 7-4: Código del proceso que rota a izquierda y derecha .....	75
Código 7-5: Código equivalente al código 7-4 .....	75
Código 8-1: Código que simula el funcionamiento del reloj de la placa Pegasus (50 MHz) .....	81
Código 8-2: Código que simula el funcionamiento del reset .....	82
Código 9-1: Declaración de un tipo enumerado para la máquina de estados .....	88
Código 9-2: Declaración de las señales de estado de tipo estados_led anteriormente declarado .....	88
Código 9-3: Proceso que obtiene el estado siguiente .....	89
Código 9-4: Proceso que actualiza y guarda el estado .....	89

Código 9-5: Proceso que proporciona la salida.....	90
Código 9-6: Declaración de tipos enumerados distintos para cada máquina de estados.....	90
Código 9-7: Código del proceso combinacional que obtiene el estado siguiente .....	93
Código 16-1: Entidad del circuito MP3 .....	123
Código 16-2: Arquitectura del circuito MP3.....	125
Código 17-1: Arquitectura del circuito de visualización del MP3 .....	131

## Índice de tablas

Tabla 6-1: Displays que se muestran según la configuración de los interruptores.....	67
Tabla 15-1:Frecuencias de las notas en distintas octavas .....	113
Tabla 16.1: Codificación de estados .....	121
Tabla 16.2: Tabla de estados siguientes y salidas .....	122
Tabla 16.3: Tabla de entradas necesarias para obtener una transición en biestables J-K .....	122
Tabla 16.4: Tabla de excitación de los necesarias para obtener una transición en biestables J-K .....	123
Tabla 18.1: Codificación de estados .....	137
Tabla 18.2: Tabla de estados siguientes y salidas .....	138
Tabla 18.3: Tabla de entradas necesarias para obtener una transición en biestables J-K .....	138
Tabla 18.4: Tabla de excitación de los necesarias para obtener una transición en biestables J-K .....	139
Tabla 20.1: Características de los componentes del circuito .....	149
Tabla 20.2: Tiempos de propagación máximos.....	150
Tabla 21.1: Tabla del estado siguiente .....	151
Tabla 21.2: Tabla de excitación del autómata .....	151
Tabla 22.1: Características de los componentes del circuito .....	153
Tabla 22.2: Tiempos de propagación máximos.....	153
Tabla 24.1: Tablas de verdad de las entradas de los biestables .....	155
Tabla 24.2: Tabla del estado siguiente .....	156
Tabla 24.3: Tabla de excitación del autómata .....	156



## 1. Introducción

Este manual es una guía práctica para aprender a diseñar circuitos digitales mediante el uso de VHDL y dispositivos lógicos programables (CPLD o FPGA). Este manual se ha desarrollado en el Departamento de Tecnología Electrónica [5] de la Universidad Rey Juan Carlos para las prácticas de la asignatura Electrónica Digital II (ED2 [8]) de la titulación de Ingeniería de Telecomunicación<sup>1</sup>. Previamente, los alumnos de esta carrera han cursado la asignatura Electrónica Digital I (ED1). En ED1 los alumnos adquirieron los conceptos básicos de la electrónica digital y realizaron diseños tanto con componentes discretos como con dispositivos lógicos programables. Las prácticas realizadas con FPGA de ED1 están guiadas en el manual de la referencia [12]. En dichas prácticas se enseña a diseñar circuitos electrónicos digitales con esquemáticos y FPGA.

Por tanto para seguir este manual con una mayor comprensión se recomienda tener conceptos básicos de los sistemas de numeración y electrónica digital: diseño con puertas lógicas, bloques combinacionales, elementos de memoria, registros y contadores. Realizar las prácticas de ED1 [12] es una buena base para seguir este manual.

Además de las prácticas de clase, en este manual se han incluido dos secciones adicionales. Por un lado, a partir de la página 103 se ha añadido una sección de circuitos digitales y analógicos en la que se incluyen dos capítulos con algunas indicaciones sobre cómo realizar la parte analógica de los circuitos. Por otro lado, a partir de la 115 se ha incluido la resolución de problemas teóricos. Estos problemas fueron ejercicios de examen durante el curso 2008-2009.

Así pues, como ya se ha dicho, en este manual aprenderemos a diseñar circuitos digitales mediante VHDL y el uso de dispositivos lógicos programables. El VHDL es un **lenguaje de descripción de hardware** que permite modelar y diseñar circuitos electrónicos digitales. Se pueden diseñar circuitos electrónicos digitales sin utilizar lenguajes de descripción de hardware. Por ejemplo, usando esquemáticos podemos diseñar un circuito electrónico digital [12]. Diseñar mediante esquemáticos es una manera más intuitiva y menos abstracta de diseñar, y por esto creemos que es la forma más recomendable para aprender a diseñar. Si no tenemos unas bases sólidas en el diseño con esquemáticos, al diseñar con VHDL podemos perder la noción de lo que estamos haciendo y pensar que estamos usando un lenguaje de programación habitual (C, Pascal, ...). No tener una idea aproximada del hardware (esquemático) que se genera a partir del código VHDL es una de las limitaciones más grandes que podemos tener como diseñadores.

Sin embargo, una vez que sabemos diseñar con esquemáticos, los lenguajes de descripción de hardware nos facilitan el diseño ya que, entre otros beneficios, nos proporcionan:

- Un nivel de abstracción mayor, ahorrándonos muchos detalles de la implementación
- Una mayor facilidad para la reutilización y hacer modificaciones
- Una mayor capacidad para manejar para manejar circuitos grandes

---

<sup>1</sup> Este plan de estudios empezó a extinguirse a partir del curso 2009-2010 con la implantación de los grados de Bolonia en el primer curso

- Posibilidad de realizar bancos de pruebas más complejos para simular la funcionalidad de nuestro diseño.

Estos beneficios los podremos apreciar si comparamos el código VHDL de los diseños propuestos con su diseño en esquemáticos. Por ejemplo, podremos comparar el contador realizado con VHDL (capítulo 6) con el contador realizado en esquemáticos (capítulo 19 de la referencia [12]).

El VHDL es un estándar del Instituto de Ingenieros Eléctricos y Electrónicos<sup>2</sup> (IEEE [13]). Existen otros lenguajes de descripción de hardware como el Verilog o el SystemC. Históricamente, el uso de VHDL o Verilog ha dividido a la comunidad de diseñadores, lo que ha provocado dificultades en el intercambio de diseños y a las empresas que fabrican herramientas informáticas de ayuda al diseño (CAD).

El VHDL es un lenguaje muy amplio que fue creado para modelar circuitos. Más tarde se empezó a utilizar para diseñar circuitos, utilizando para ello sólo un conjunto reducido del VHDL, lo que se llama **VHDL para síntesis** [14]. La transformación de un circuito descrito en VHDL a su esquema en puertas lógicas y biestables se llama síntesis. Esta síntesis la realizan automáticamente las herramientas CAD, gracias a esto los diseñadores ahorrarán mucho tiempo de diseño. En este manual no aprenderemos a utilizar la totalidad del VHDL, sino que usaremos un conjunto restringido del lenguaje orientado a síntesis. Lo que es más, tampoco usaremos todo el conjunto de VHDL para síntesis, sino que utilizaremos lo que vayamos necesitando. Por suerte o por desgracia, en VHDL una cosa se puede describir de muchas maneras distintas. Por lo general, en este manual aprenderemos sólo una de ellas.

Por último, con el fin de que este manual pueda llegar al mayor número de personas y con el objetivo de generar material educativo abierto, hemos publicado este manual bajo licencia *Creative Commons* [3] que permite copiarlo y distribuirlo. Esperamos que disfrutes de su lectura y que te ayude a aprender a diseñar circuitos electrónicos digitales. Para ir mejorando el manual, agradeceremos la comunicación de comentarios, sugerencias y correcciones a las direcciones de correo electrónico de los autores.

[felipe.machado@urjc.es](mailto:felipe.machado@urjc.es) y [susana.borromeo@urjc.es](mailto:susana.borromeo@urjc.es).

---

<sup>2</sup> Institute of Electrical and Electronics Engineers

## 2. Encender un LED

En las prácticas de este manual realizaremos diseños bastante más complejos que los que hicimos en Electrónica Digital I [12]. Por un lado porque ya sabemos más de electrónica y por otro lado porque diseñaremos con VHDL. El VHDL es un lenguaje de descripción de hardware que nos permitirá realizar diseños avanzados de manera más rápida y eficiente que cuando usábamos los esquemáticos.

Como ya vimos en las prácticas de Electrónica Digital I, siempre que se introduzca una novedad en el diseño debemos realizar un diseño de prueba lo más sencillo posible para ver si todo funciona bien. En nuestro caso, en esta primera práctica queremos probar el proceso de diseño en VHDL, así que vamos a realizar un diseño VHDL que sea muy sencillo para probar que todo lo proceso va bien. Por tanto, nuestro diseño de prueba será encender un LED describiendo el circuito en VHDL en vez de en esquemáticos.

Esta práctica nos servirá para repasar el entorno de diseño de *Xilinx* [25]: el *ISE WebPack*<sup>3</sup> [15]. En el resto de las prácticas no se hará referencia al entorno de desarrollo. Así que si tienes dudas de cómo implementar el circuito en la FPGA, vuelve a esta práctica o repasa las prácticas del año pasado [12].

---

### 2.1. Tarjetas Pegasus y Basys

La tarjeta que utilizaremos en prácticas es la *Pegasus* [19] de la empresa *Digilent* [6]. Esta tarjeta ya no se produce, pero se podrá utilizar sin mucha dificultad cualquier otro tipo de placas que tengan características similares, como la *Basys* [2] o la *Nexys*,...

Como hay algunas diferencias en la programación de la *Pegasus* y la *Basys*, en los siguientes subapartados se explicarán cada una de ellas.

#### 2.1.1. La tarjeta Pegasus

La tarjeta *Pegasus* contiene una FPGA de *Xilinx* de modelo *Spartan-II XC2S50* ó *XC2S200* con encapsulado *PQ208*. Estas FPGA tienen respectivamente 50000 ó 200000 puertas lógicas equivalentes y 140 pines disponibles para el usuario. Con esta capacidad podemos hacer diseños digitales bastante grandes.

La placa *Pegasus* contiene distintos periféricos que nos facilitarán la tarea del diseño, como por ejemplo: 8 LED, 4 displays de 7 segmentos, 4 botones, 8 interruptores, un puerto PS/2 (para ratón o teclado), un puerto VGA (para conectar un monitor), y varios puertos de expansión. En la figura 2.1 se muestra la placa *Pegasus*, y en la que se señalan varios de sus componentes. La foto de la placa no coincide exactamente con la versión de la placa que tenemos en el laboratorio y hay algunos componentes que no están en el mismo lugar, aunque sí todos los que se han señalado.

---

<sup>3</sup> Herramienta gratuita que se puede descargar en:

<http://www.xilinx.com/webpack/classics/wpclassic/index.htm>

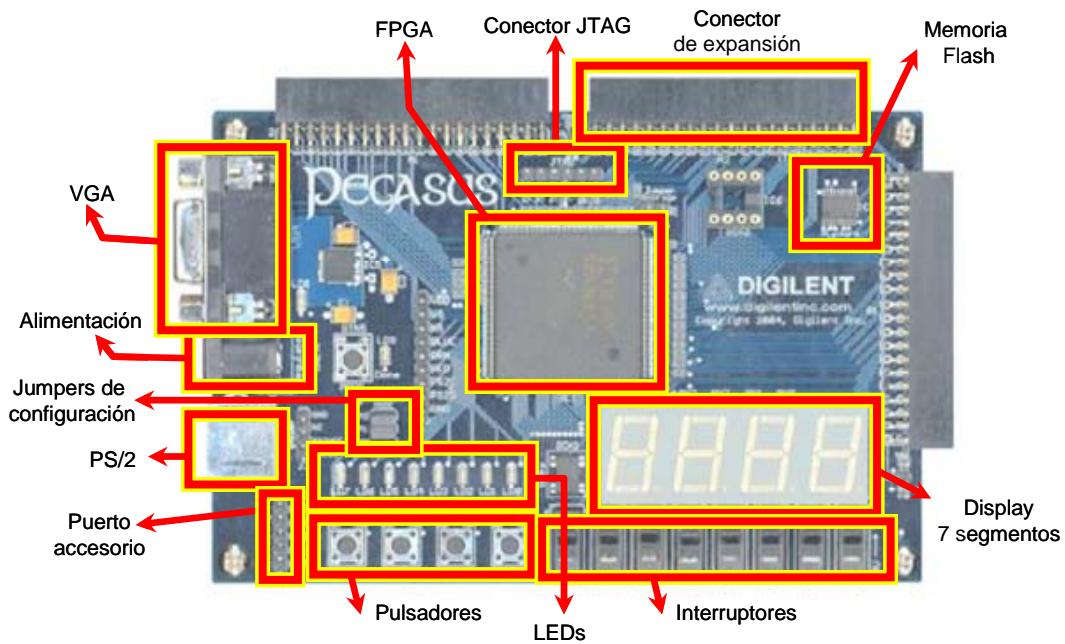


Figura 2.1: Placa Pegasus de Digilent

A medida que vayamos avanzando con la práctica iremos aprendiendo la funcionalidad de la placa. Aún así, si tienes curiosidad, en la página web de *Digilent* puedes consultar el manual de la placa en inglés [19].

### 2.1.2. La tarjeta Basys

La tarjeta *Basys* contiene una FPGA de Xilinx de modelo *Spartan-3E XC3S100E* ó *XC3S250E* con encapsulado TQ144. Estas FPGA tienen respectivamente 100000 ó 250000 puertas lógicas equivalentes y 108 pines disponibles para el usuario.

La placa *Basys* tiene muchos periféricos similares a la *Pegasus*, por ejemplo: 8 LED, 4 displays de 7 segmentos, 4 pulsadores, 8 interruptores, un puerto PS/2 y un puerto VGA. Los puertos de expansión son diferentes a la *Pegasus*, la tarjeta *Basys* tiene 4 puertos de expansión de tipo PMOD, con los que podemos conectar una gran variedad de periféricos disponibles<sup>4</sup> o incluso podemos hacernoslos nosotros mismos. La *Pegasus* sólo tiene un puerto PMOD. En la figura 2.2 se muestra la revisión E de la placa *Basys*, en ella se señalan varios de sus componentes.

<sup>4</sup> Los conectores PMOD son muy variados: pantallas de cristal líquido, circuitos control de motores, antenas, joysticks, amplificadores de audio, micrófonos, etc. Se pueden ver en <http://www.digilentinc.com/Products/Catalog.cfm?NavPath=2,401&Cat=9>

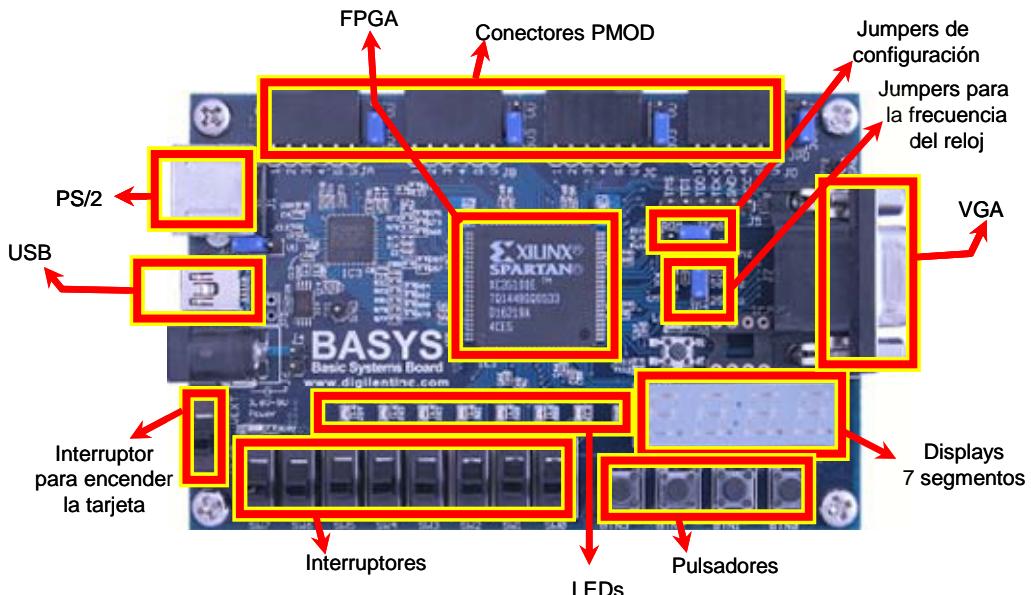


Figura 2.2: Placa Basys de Digilent

## 2.2. Cómo encender un LED

Del año pasado ya sabemos encender un LED. Ya hemos visto que las placas *Pegasus* y *Basys* tienen 8 LED disponibles para utilizar, cada uno de estos LED van conectados a distintos pines de la FPGA. La conexión de los LED se muestra en la figura 2.3. En ella se indican qué pines de la FPGA están conectados a los LED. Por ejemplo el LED número 0 de la *Pegasus* está conectado al pin 46. Para facilitar esta identificación, la placa *Pegasus* lleva impresos la identificación del LED y el pin asociado de la FPGA (en la *Basys* no<sup>5</sup>). Así, podemos ver cómo al lado del LED 0 (LD0) de la *Pegasus* está el número 46 entre paréntesis. Podemos apreciar que esto se cumple para casi todos los componentes: pulsadores, interruptores, .... Esto no ocurre en la *Basys* y para ella tenemos que consultar su manual de referencia.

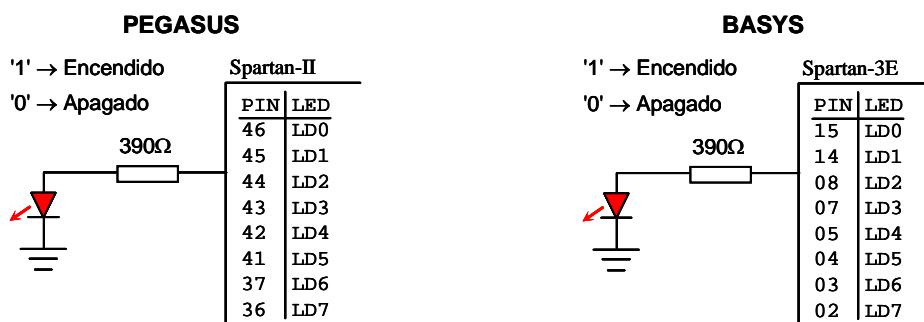


Figura 2.3: Conexión de los LED de la placa Pegasus (izquierda) y la Basys (derecha)

Viendo el esquema eléctrico de la figura 2.3 podemos deducir que si ponemos un '1' lógico en el pin 46 de la FPGA de la *Pegasus* encenderemos el LED, ya que con un '1' lógico la FPGA pondrá una tensión de 3,3 voltios en ese pin (que es el nivel de tensión de ese pin de la FPGA). Por el contrario, si ponemos un '0' lógico en el pin 45, mantendremos apagado el LED 1 (LD1), puesto que no circulará corriente por él.

<sup>5</sup> Ten cuidado porque la *Basys* viene con un dibujo con la correspondencia con los pines, y algunos de ellos están mal. Es más seguro consultarlos en el manual de referencia.

Así pues, éste será el objetivo del primer ejercicio, poner un '1' en el pin 46 y un '0' en el pin 45 (para la Basys los pondremos en los pines 15 y 14 respectivamente). Si después de programar la FPGA el LED 0 se enciende y el LED 1 se apaga habremos cumplido el objetivo. Con el resto de pines que van a los otros LED no haremos nada, en teoría, haciéndolo así no deberían lucir, ya que por defecto la FPGA pone resistencias de *pull-down* en los pines no configurados, y una resistencia de *pull-down* pone la salida a tierra.

## 2.3. Diseño del circuito

El diseño del circuito lo haremos con la herramienta *ISE-Foundation* o con *ISE-WebPack* de Xilinx, esta última es la versión gratuita que se puede descargar en la página web de Xilinx. Esta práctica está referida a la versión 9.2, debido a que existen nuevas versiones, puedes descargar esta versión en: [http://www.xilinx.com/ise/logic\\_design\\_prod/classics.htm](http://www.xilinx.com/ise/logic_design_prod/classics.htm)

Para arrancar el programa pincharemos en el icono de *Xilinx* o bien, desde:

*Inicio*→*Todos los programas*→*Xilinx ISE 9.2i*→*Project Navigator*



Xilinx ISE 9.2i

Nos puede aparecer una ventana con el *Tip of the day* que son indicaciones que hace la herramientas cada vez que la arrancamos. Si las leemos habitualmente podemos ir aprendiendo poco a poco. Pinchamos en *Ok*, con lo que se cierra dicha ventana.

Normalmente la herramienta abre el último proyecto que se ha realizado. Así que lo cerramos pinchando en: *File*→*Close Project*.

Para empezar a crear nuestro nuevo proyecto, pinchamos en *File*→*New Project...* y nos saldrá la ventana *New Project Wizard – Create New Project* como la mostrada en la figura 2.4. En ella pondremos el nombre del proyecto, que lo llamaremos *led0*, indicamos la ruta donde guardaremos el proyecto, que será `C:/practicas/ed2/tunombre`. Respecto al nombre y a la ruta, es conveniente:

- **No trabajar desde un dispositivo de memoria USB<sup>6</sup>.**
- **No incluir espacios en la ruta**, por tanto, esto incluye **no trabajar en el "Escritorio" ni en "Mis documentos"**
- No incluir en la ruta o en el nombre **acentos ni eñes**, ni caracteres extraños, ni nombres muy largos, lo más conveniente es limitarse a caracteres alfanuméricos, y usar el guión bajo en vez del espacio.

Para el último recuadro de la ventana, donde pone *Top-Level Source Type* seleccionaremos *HDL*, ya que nuestro diseño lo haremos mediante lenguajes de descripción de hardware<sup>7</sup>.

---

<sup>6</sup> Si estás trabajando en el laboratorio, al terminar la sesión no olvides copiar los ficheros con extensión `.vhd` y `.ucf` en tu memoria USB o en tu correo electrónico (ocupan muy poco). De lo contrario, el próximo día puede ser que los ficheros hayan desaparecido.

<sup>7</sup> HDL: del inglés: *Hardware Description Language*

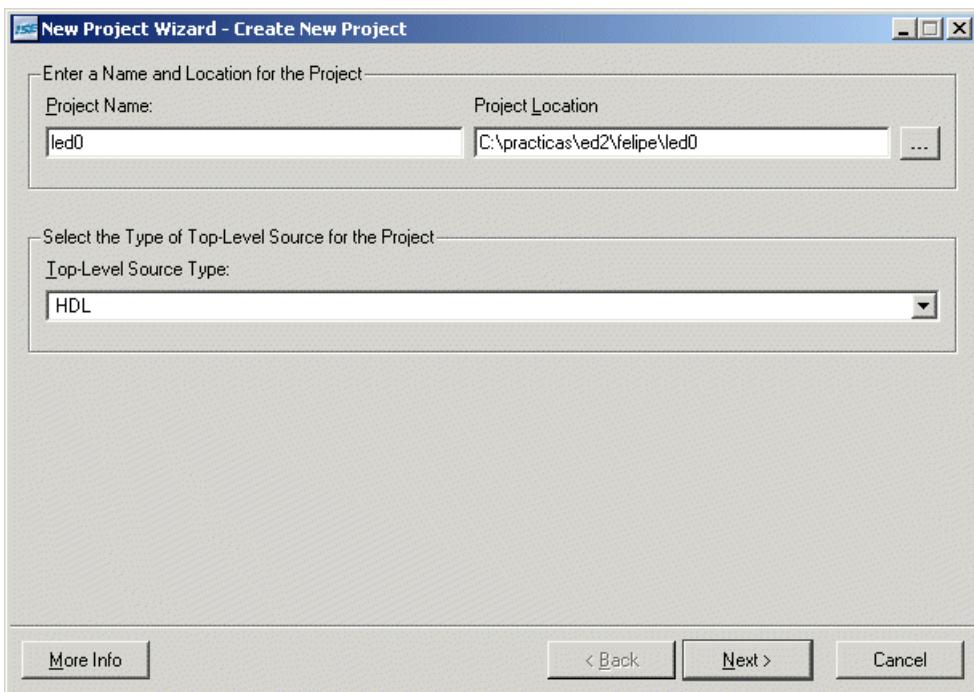


Figura 2.4: Ventana para la creación de nuevo proyecto

Una vez que hemos rellenado los tres recuadros pinchamos en *Next*. Ahora nos aparece la ventana de selección del dispositivo (figura 2.7). En *Product Category* ponemos *All*. En la familia ponemos:

- *Spartan2* si tenemos la *Pegasus*
- *Spartan3E* si usamos la *Basys*

Los siguientes datos los podemos obtener a partir de la observación del texto del encapsulado de la FPGA, que se muestran en la figura 2.5 para la *Pegasus* y la figura 2.6 para la *Basys*. Hay que observar los valores que tiene la FPGA de nuestra placa, ya que por ejemplo, en la *Pegasys* en algunas el tipo es *xc2s200* y en otras es *xc2s50*. Lo mismo sucede con la velocidad, en algunas es 5 y otras 6.

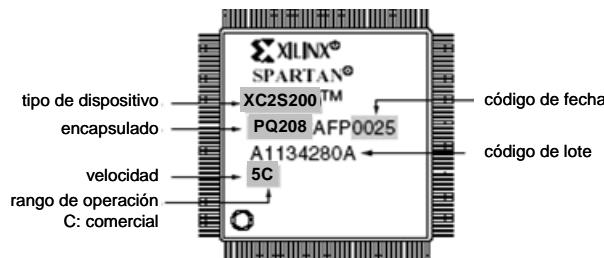


Figura 2.5: Interpretación del texto del encapsulado de la FPGA de la *Pegasus*

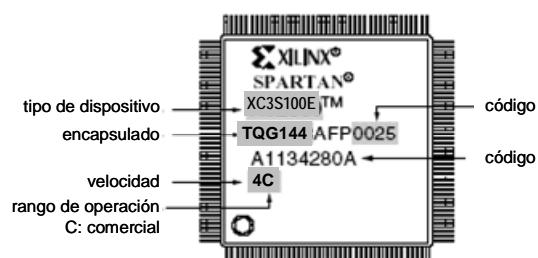


Figura 2.6: Interpretación del texto del encapsulado de la FPGA de la *Basys*

Para la selección de la herramienta de síntesis no tenemos alternativa, y para el simulador en esta práctica no es importante y lo dejamos en *ISE Simulator (VHDL/Verilog)*. En el lenguaje de descripción de hardware preferido (*Preferred Language*) se deja el *VHDL*, que es el que usaremos. Al terminar pinchamos en *Next*.

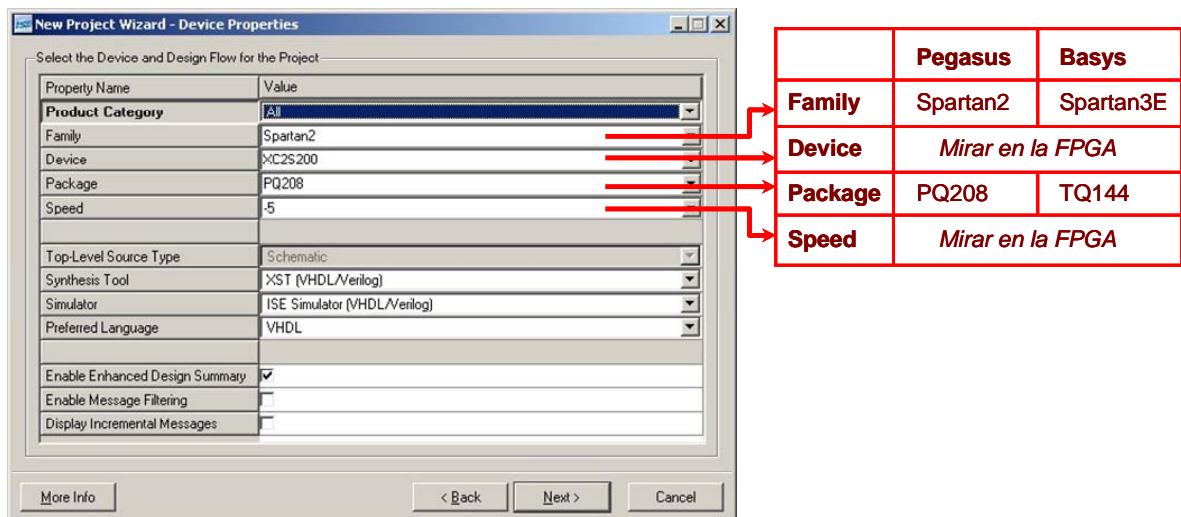


Figura 2.7: Ventana para la selección del dispositivo del nuevo proyecto

En las dos siguientes ventanas pinchamos en *Next* sin rellenar nada y en la última pinchamos en *Finish*.

Ya tenemos el proyecto creado y ahora nos disponemos a crear nuestro diseño. Para ello creamos una nueva fuente (fichero) para el proyecto pinchando en *Project→New Source*.

Esto hará aparecer una nueva ventana que nos pedirá que seleccionemos el tipo de fuente que queremos crear. Como estamos trabajando en VHDL, seleccionamos *VHDL Module*, y nombramos al fichero *led0*. Este paso se muestra en la figura 2.8. Posteriormente pinchamos en *Next..*.

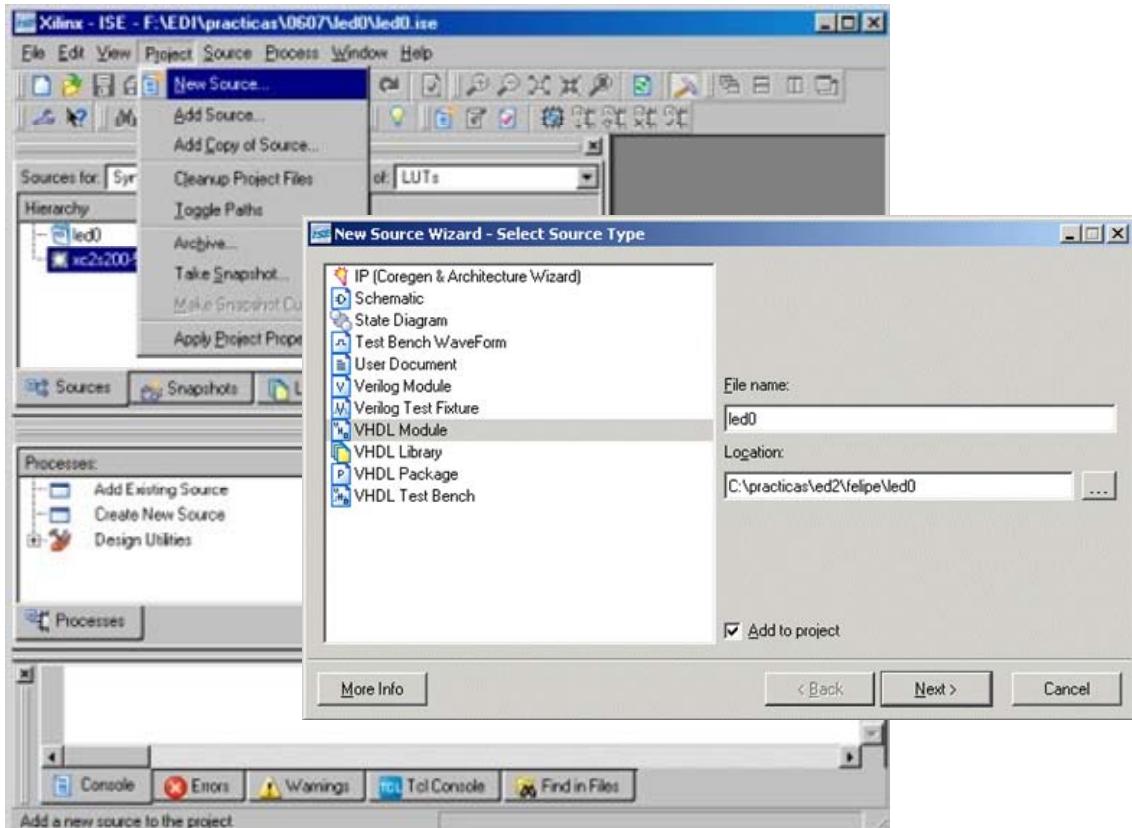


Figura 2.8: Selección del tipo de la nueva fuente que vamos a crear

Ahora nos saldrá una ventana como la mostrada en la figura 2.9. Por defecto, el nombre de la arquitectura (*Architecture Name*) es *Behavioral* (comportamental). Éste se puede cambiar, pero por ahora, lo dejamos así. En esta ventana podemos indicar los puertos de nuestra entidad. En la figura 2.9 se han creado dos puertos de salida: led0 y led1. Otra alternativa es crearlos nosotros directamente en VHDL. Para terminar pinchamos *Next* y luego en *Finish*.

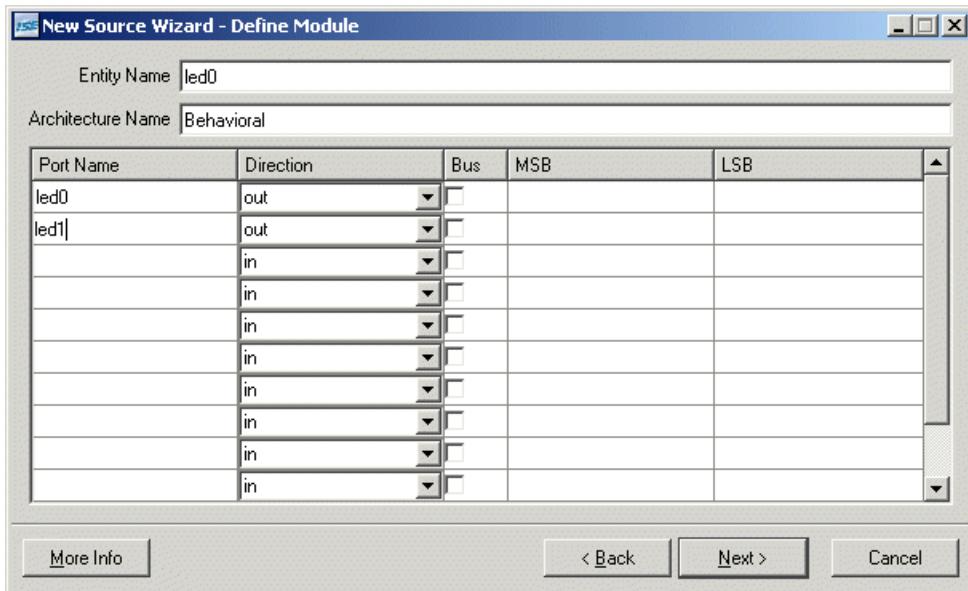


Figura 2.9: Definición de los puertos

Si al terminar el proceso la herramienta nos muestra la siguiente imagen (figura 2.10), debemos seleccionar el fichero que hemos creado (*led0.vhd*), ya sea pinchando en su pestaña, o ya sea pinchando en *Window→led0.vhd*

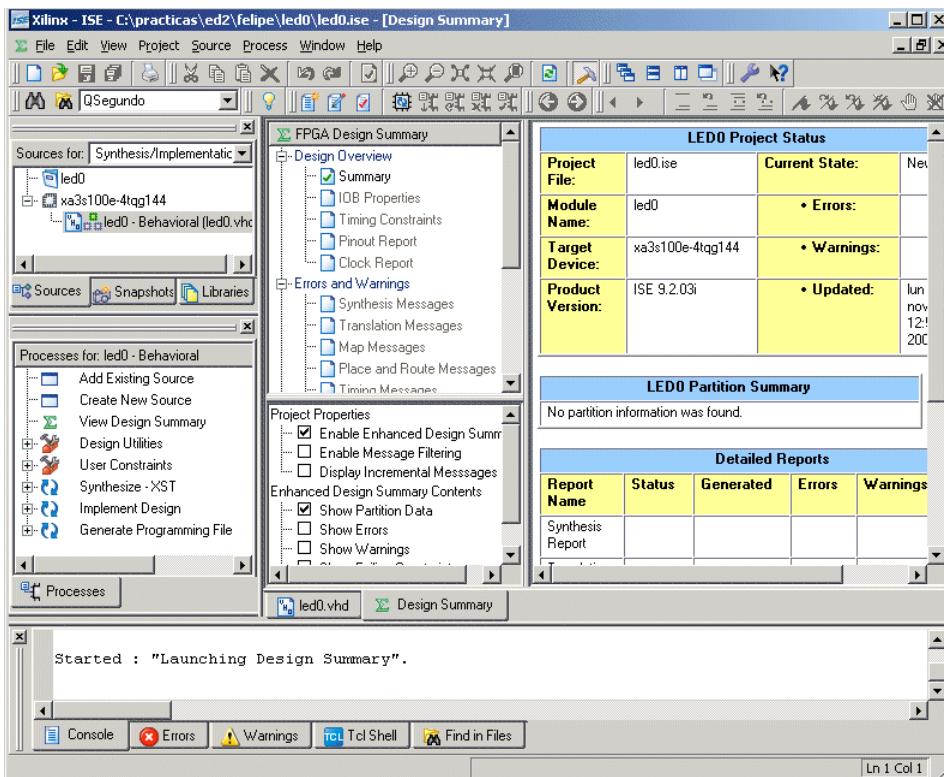


Figura 2.10: Apariencia de la herramienta al añadir la nueva fuente led0.vhd

Como podemos ver en la figura 2.11, el fichero VHDL contiene una cabecera añadida automáticamente por la herramienta. Esta cabecera consiste en comentarios que el diseñador debería de llenar para identificar y ayudar a entender el diseño. Para este ejemplo no es necesario, pero es interesante para diseños complejos. Observa que cada línea de la cabecera comienza por dos guiones "--" que es la manera de comentar en VHDL. Esto es, la herramienta no analiza las líneas que contienen dos guiones y puedes poner cualquier comentario.

```

8 -- Project Name:
9 -- Target Devices:
10 -- Tool versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22 use IEEE.STD_LOGIC_UNSIGNED.ALL;
23 use IEEE.STD_LOGIC_ARITH.ALL;
24
25 ---- Uncomment the following library declaration if instantiating
26 ---- any Xilinx primitives in this code.
27 --library UNISIM;
28 --use UNISIM.VComponents.all;
29
30 entity led0 is
31     Port ( led0 : out STD_LOGIC;
32             led1 : out STD_LOGIC);
33 end led0;
34
35 architecture Behavioral of led0 is
36 begin
37
38 end Behavioral;

```

Figura 2.11: Apariencia de la herramienta para empezar a realizar la descripción VHDL

Después de la cabecera están las referencias a las bibliotecas del IEEE<sup>8</sup>, que son necesarias para poder utilizar los tipos de datos que emplearemos. Para nuestro caso, nos basta con utilizar la biblioteca IEEE.STD\_LOGIC\_1164. Las otras bibliotecas no son necesarias por ahora.

Luego tenemos unos comentarios que podemos eliminar. Después de éstos, ya tenemos la **entidad**, que ha sido creada por el ISE con la información de los puertos que le hemos dado, que son dos puertos de salida: led0 y led1.

En VHDL, la entidad define las entradas y salidas del circuito. Y no hace referencia a la estructura o funcionalidad interna. Es como definir una caja negra. En nuestro ejemplo, la entidad se puede representar como muestra la figura 2.12.

<sup>8</sup> El IEEE (Institute of Electrical and Electronics Engineers) es la asociación que ha estandarizado el VHDL.

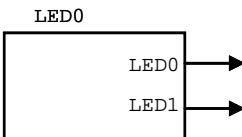


Figura 2.12: "Caja negra" que define la entidad de nuestro circuito

El ISE ha creado los puertos de tipo `STD_LOGIC`, que es un tipo de dato parecido al `BIT`, pero que no sólo tiene los valores binarios '0' y '1', sino que puede tomar otros valores como:

- 'U': *Unassigned*. La señal no ha recibido valor
- 'X': La señal ha recibido valores distintos
- 'Z': Alta impedancia
- '-': No importa (*don't care*)

Hay más valores posibles que no se han incluido para no complicar innecesariamente la explicación.

Ahora queda describir la arquitectura. En VHDL, la arquitectura describe la estructura, el funcionamiento o el comportamiento de un circuito.

En nuestro ejemplo, nosotros queremos asignar a un puerto el valor '1' y al otro el valor '0'. Esto se realiza con una sentencia concurrente para cada LED. El código 2-1 muestra cómo se realiza en VHDL. El VHDL no distingue entre mayúsculas y minúsculas.

```

architecture BEHAVIORAL of LED0 is
begin
    led0 <= '1';          -- Encendemos el LED0
    led1 <= '0';          -- Apagamos el LED0
end BEHAVIORAL;

```

Código 2-1: Arquitectura que enciende un LED y apaga otro

Si nos fijamos en el código 2-1, vemos que el nombre de la arquitectura (`BEHAVIORAL`) indica el tipo de descripción: en este caso la arquitectura explica el comportamiento del circuito. En realidad, aquí se puede poner el nombre que uno quiera, aunque existe una convención de indicar el tipo de descripción:

- BEHAVIORAL (O COMPORTAMENTAL)
- STRUCTURAL (O ESTRUCTURAL)
- FUNCTIONAL (O FUNCIONAL)
- DATAFLOW (O FLUJO\_DE\_DATOS)

Ya iremos viendo qué significan estos descriptores.

Después del nombre de la arquitectura se indica a qué entidad pertenece dicha arquitectura. En nuestro ejemplo: "of `LED0`". Indica que pertenece a la entidad `LED0`. Aunque no es obligatorio, la entidad y arquitectura se pueden describir en el mismo fichero.

Por último, a partir del `BEGIN` se incluyen las sentencias concurrentes de nuestro circuito. En este caso, son dos asignaciones a dos valores constantes. La asignación de señales se realiza con el operador "`<=`".

Ahora incluye las sentencias del código 2-1. Para mayor claridad de código, *indent<sup>9</sup>* las sentencias concurrentes con dos espacios a la derecha. Una vez que hemos terminado con el diseño de ejemplo, lo grabamos (*File→Save*).

## 2.4. Síntesis e implementación del circuito

Ahora selecciona la ventana de *procesos* y la ventana de *fuentes* (*Sources*). En la ventana de *fuentes* verifica que el componente y el fichero que hemos creado está seleccionado: *led0-Behavioral(led0.vhd)*. Pincha en él una vez si no estuviese seleccionado.

Aunque a los puertos de salida les hayamos dado unos nombres referidos a los LED que vamos a utilizar, la herramienta no sabe qué pines queremos utilizar. Para indicarle a la FPGA qué pines vamos a utilizar y con qué puertos de nuestro diseño los vamos a conectar deberemos lanzar la herramienta *PACE*. Esto lo hacemos desde la ventana de *procesos*, en ella despliega la sección que dice *User Constraints*, pincha dos veces en *Assign Package Pins* (figura 2.13). Posteriormente nos saldrá una ventana de aviso que nos indica que para este proceso se requiere añadir al proyecto un fichero del tipo *UCF*. Este tipo de ficheros son los que se usan para definir las conexiones de los pines de la FPGA con los puertos de nuestro diseño, y por tanto pinchamos en *Yes*.

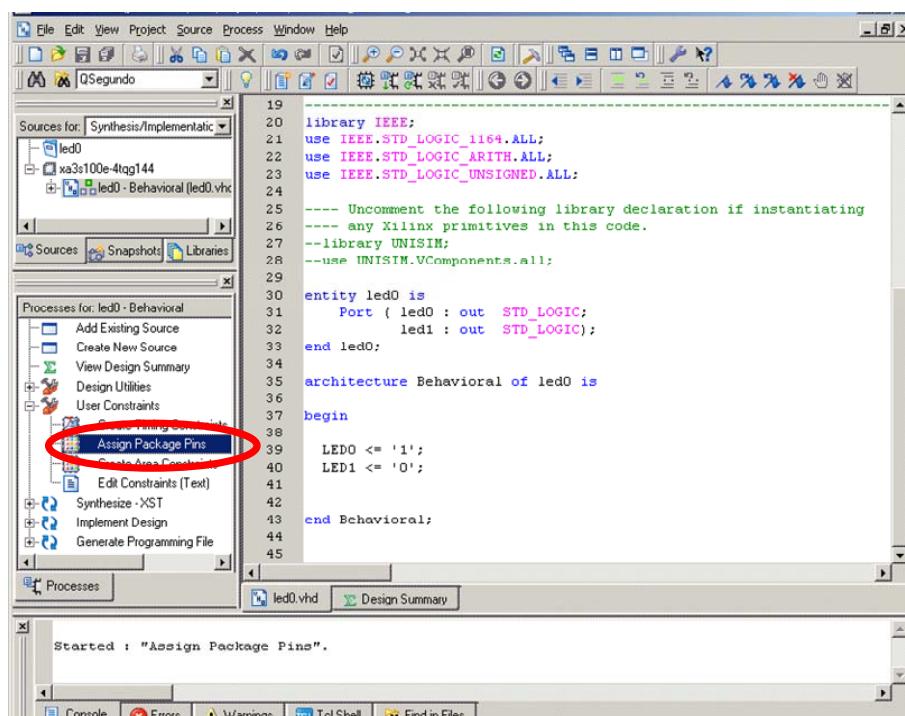


Figura 2.13: Selección para la asignación de pines

Ahora nos aparecerá la herramienta *Xilinx PACE*. En ella podemos distinguir tres subventanas, la de la derecha, tiene 2 pestañas en su parte inferior, en una se muestra la arquitectura de la FPGA y en la otra la asignación de los pines (ver figura 2.14).

En la subventana de la parte inferior izquierda hay una tabla con los nombres de nuestros puertos: *led0* y *led1*. En una columna se indica la dirección de los puertos (ambos de

<sup>9</sup> *Indentar* es insertar espacios o tabuladores para mover un bloque de código a la derecha y facilitar la comprensión del código. El término *indentar* no está reconocido en la RAE y para ser correctos habría que utilizar el término "sangrado"

salida: *output*). Y en la siguiente columna (*Loc*) debemos indicar el pin de la FPGA con el que queremos unir nuestros puertos. Ten cuidado porque aquí hay un fallo de la herramienta, ya que si pinchamos en las celdas de dicha columna aparece un menú desplegable con los bancos de la FPGA (los pines se organizan en bancos), pero en estas celdas hay que indicar los pines y no los bancos. Así que pinchamos en dichas celdas y ponemos nosotros mismos el pin que queremos conectar. Así, si usamos la *Pegasus*, para el puerto `led0` conectamos el pin 46, para ello escribimos `P46` (el mismo número que indica la placa añadiéndole una `P`). Para el puerto `LED1` le asignamos el pin `P45`. Si usamos la placa *Basys*, tendríamos que poner los pines `P15` y `P14`, respectivamente.

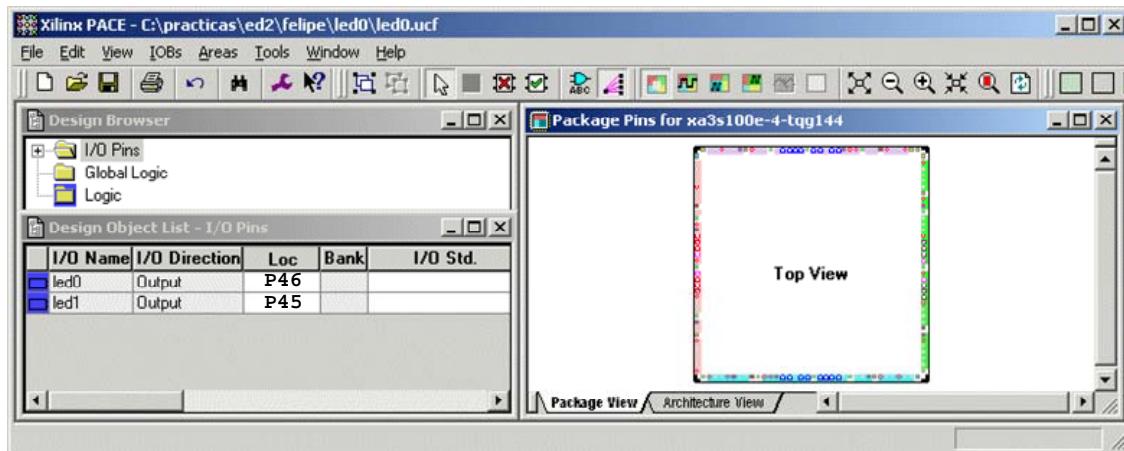


Figura 2.14: Herramienta PACE para la asignación de pines

Le damos a guardar y nos saldrá una ventana que nos pregunta por el formato del fichero que guardará esta información. Aunque nos es indiferente, seleccionamos la primera opción "XST Default <>", pinchamos en *OK*, cerramos la herramienta PACE y volvemos al ISE.

En la ventana de procesos, tenemos ordenados los pasos que debemos tomar: "*Synthesize-XST*", "*Implement Design*" y "*Generate Programming File*". Así que con este orden, vamos a ir pinchando dos veces en cada uno de ellos hasta que veamos que se pone una signo verde de correcto en ellos. Cuando se ponga en verde, pinchamos en el siguiente (puede salir en amarillo si indicando que hay advertencias -*warnings*- que a veces no son importantes). En realidad, si se ejecuta el último paso sin haber ejecutado los anteriores, la herramienta realiza todo el proceso.

Cuando tengamos todos los pasos correctamente ejecutados, la subventana de procesos debería mostrar el aspecto de la figura 2.15. Si es así, ya estamos en disposición de programar la FPGA. Puede ser que aparezca algún *warning* y que el proceso de síntesis tenga un símbolo amarillo triangular. Los *warnings* se pueden ver en la ventana de *design summary*. De todos modos, en muchas ocasiones estas advertencias no son relevantes.

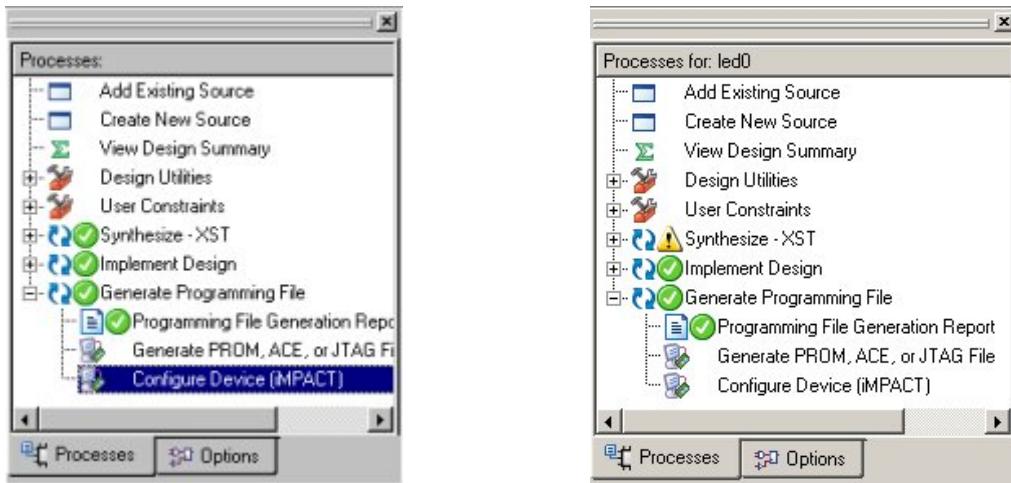


Figura 2.15: Aspecto de la subventana de procesos una vez que se ha sintetizado e implementado el diseño correctamente. En la ventana de la derecha, en el proceso de síntesis ha habido advertencias (warnings) que no siempre son importantes, y por eso tiene un símbolo triangular de color amarillo

## 2.5. Programación de la FPGA

El procedimiento para la programación de la FPGA es diferente según la tarjeta que usamos. En los siguientes apartados se explican ambos procedimientos.

### 2.5.1. Programación de la tarjeta Pegasus

Si hemos llegado hasta aquí, ya habremos generado el fichero de programación de la FPGA (que tiene extensión .bit). Ahora veremos los pasos que hay que dar para programar la placa *Pegasus* (la *Basys* requiere unos pasos diferentes).

Lo primero que tenemos que hacer es quitar los tres *jumpers* de configuración: M0 , M1 y M2 del conector J4, (ver situación en la figura 2.1). Para no perder las caperuzas conéctalos a un sólo conector dejando el otro al aire como se muestra a la derecha de la figura 2.16.

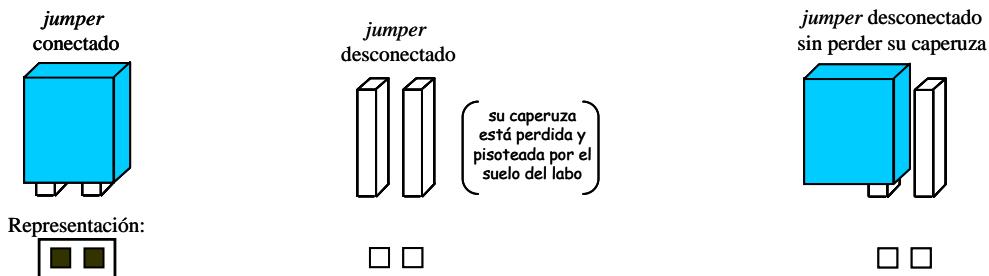


Figura 2.16: Jumpers y su conexión

Cuando estos *jumpers* están conectados, la FPGA se programa desde la memoria *flash* de la placa, en la que se guarda un fichero de configuración para la FPGA. Las *Spartan* son un tipo de FPGA volátil, esto es, requiere estar alimentada a una fuente de tensión para mantener su configuración. Por tanto, cuando la apagamos se borra lo que hayamos programado. Para evitar tener que programar la FPGA desde el ordenador cada vez que encendamos nuestro sistema electrónico, se utiliza una memoria *flash* que la programa automáticamente. Si esto no fuese posible, este tipo de FPGA perdería casi toda su utilidad, ya que no serían autónomas y prácticamente sólo valdrían para hacer pruebas en el laboratorio.

Como nosotros estamos haciendo pruebas, queremos programar la FPGA desde el ordenador y no desde la memoria *flash*, así que desconectamos los tres *jumpers*.

Ahora hay que conectar el cable JTAG al puerto paralelo del ordenador y conectar el otro extremo del cable al conector JTAG de la placa (conector J2, ver figura 2.1). No debes conectarlo de cualquier manera, de las dos posibilidades que hay, hazlo de forma que los nombres de las señales de la placa coincidan con los nombres de las señales del cable. Esto es: TMS con TMS, TDI con TDI, ... , y VDD con VDD o con VCC (éste es el único caso en que pueden no coincidir exactamente, depende del conector).

A continuación conectamos el cable de alimentación de la placa y lo enchufamos a la red eléctrica<sup>10</sup>. Al enchufar la placa verás que se enciende el LED LD8.

Ya tenemos todo preparado para programar la FPGA y probar nuestro diseño. Volvemos al ordenador, al programa ISE. Dentro de "Generate Programming File" desplegamos el menú pinchando el cuadradito con el signo + dentro:  . Y pinchamos dos veces en "Configure Device (iMPACT)".

Esto arrancará la herramienta *iMPACT*, que nos mostrará un ventana en la que se ofrecen varias opciones, dejamos la opción que hay por defecto "*configure devices using Boundary-Scan (JTAG)*" y "*Automatically connect to a cable and identify Boundary-Scan chain*". Aceptamos pinchando en *Finish*.

Lo que hemos hecho es decirle a la herramienta que identifique los componentes que hay en la placa por medio del protocolo *Boundary-Scan* (JTAG). Después de esto la herramienta procede a identificar a la placa y debería encontrar dos componentes: la FPGA y la memoria *flash*. La FPGA la identifica como *xc2s200* ó *xc2s50* y la memoria como *xcf02s*. A continuación nos pedirá los ficheros de programación para cada uno de los componentes (figura 2.17).

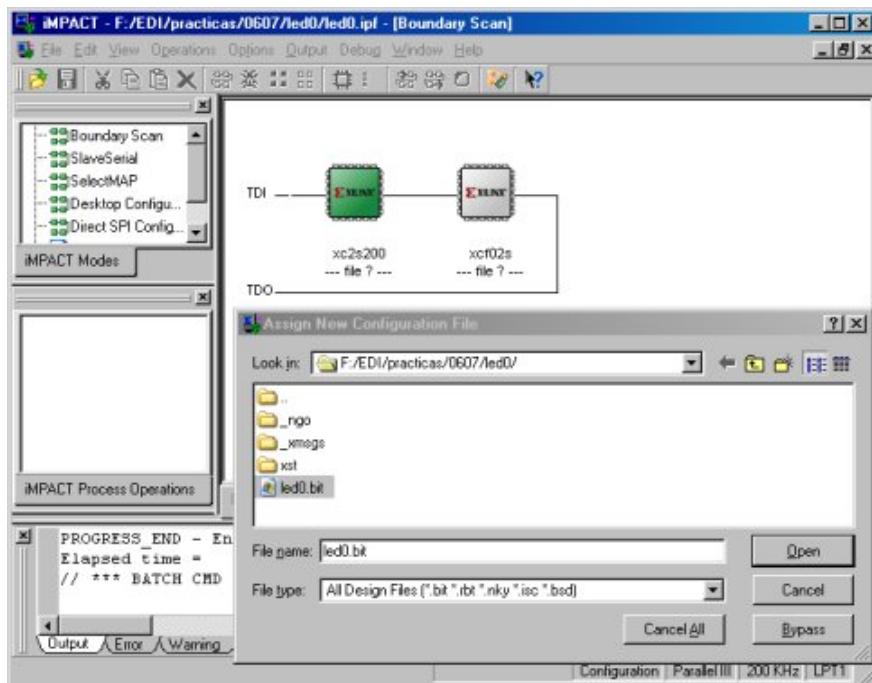


Figura 2.17: *iMPACT*: componentes detectados y asignación de un fichero de configuración

<sup>10</sup> Cuando estés en el laboratorio y lo desenchufes, fíjate en que no se quede enchufado el adaptador de la clavija plana americana a clavija redonda europea

Para la FPGA hay que asignar el fichero `led0.bit`, sabremos que está asignando el fichero a la FPGA porque ésta tendrá color verde y además la extensión del fichero es `.bit` (para las memorias es otro). Seleccionamos el fichero y pinchamos en *Open*. Inmediatamente nos saldrá una advertencia (*Warning*) que habla sobre el *Startup Clock*, esto no nos afecta y pulsamos en *Ok*.

Ahora nos pide el fichero de configuración de la memoria. Como no vamos a programarla, pulsamos en *Cancel*.

En la ventana central del *iMPACT* tenemos los dos componentes. Pinchando en cualquiera de ellos, los seleccionamos y se pondrán de color verde. Si pinchamos en la FPGA (`xc2s200` ó `xc2s50`) con el botón derecho del ratón, nos saldrá un menú. Puede pasar que al pinchar no salga y que aparezca de nuevo la ventana de la figura 2.17, eso es que se ha quedado el puntero del ratón enganchado; para evitarlo, pincha con el botón izquierdo en cualquier parte de la ventana fuera de los símbolos de los chips y vuelve a intentarlo. Ahora, en el menú que aparece podemos asignar un fichero de configuración diferente o programar la FPGA (figura 2.18). Y esto es lo que vamos hacer, pinchamos en *Program*, y le damos a *Ok* a la siguiente ventana sin cambiar nada (si diésemos a *Verify* daría un error de programación).

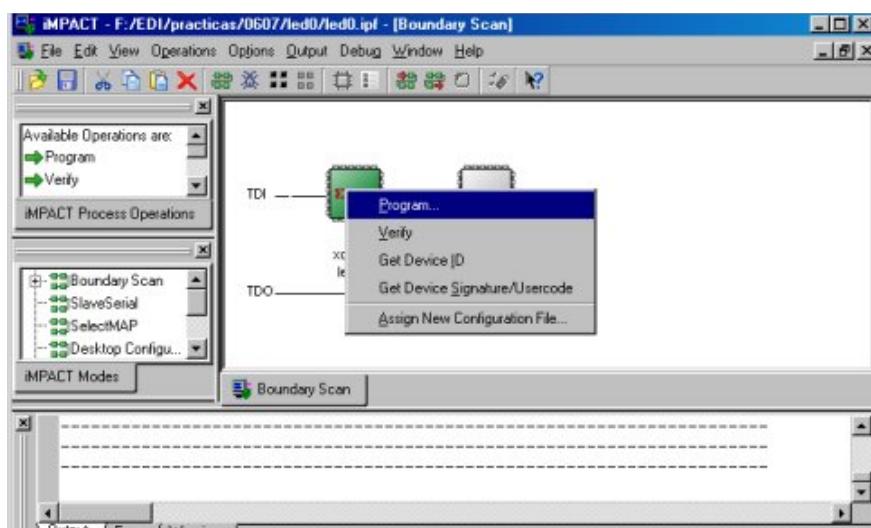


Figura 2.18: Orden para programar la FPGA

Después de la orden de programar la FPGA, aparece un cuadro que muestra la evolución y, si todo ha salido bien, la herramienta nos avisará de que hemos tenido éxito en la programación mostrando un aviso como el de la figura 2.19.

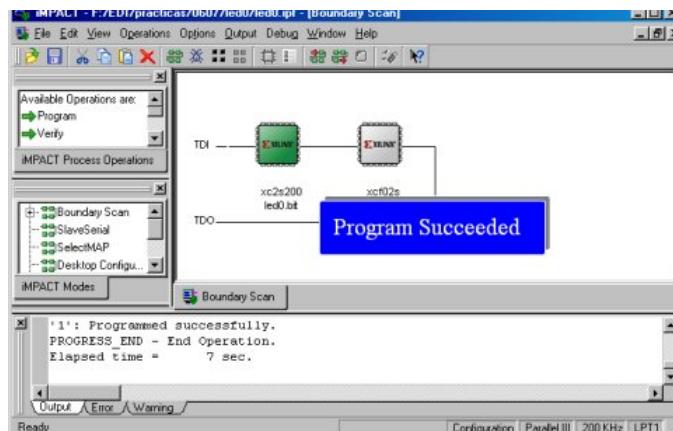


Figura 2.19: Programación realizada con éxito

Ahora miramos la placa y debería haber tres LED encendidos:

- LD8: que indica que la placa está encendida
- LD9: que indica que se ha programado la FPGA
- LD0: que es el LED que hacemos encender nosotros con el diseño de la FPGA

El LED LD1 deberá estar apagado, pues lo hemos puesto a un '0' lógico. Y el resto de LED también estarán apagados debido a las resistencias de *pull-down* que la FPGA pone en los puertos sin conectar.

## 2.5.2. Programación de la tarjeta Basys

Aunque la tarjeta *Basys* se puede programar con el conector JTAG, viene con un puerto USB que permite programarla de manera más cómoda. El único inconveniente es que hay que instalar el programa gratuito *Adept* [1].

Antes de conectar nada, lo primero que tenemos que hacer es situar el *jumper* de configuración en la posición *JTAG* (en la parte derecha). El *jumper* de configuración lo puedes localizar en la figura 2.2. Pon la caperuza azul en los dos pines de la derecha, donde dice *JTAG*. Esto es para que la FPGA se programe desde el JTAG y no se programe a partir de lo que haya grabado en la memoria ROM (en realidad es una memoria *flash*). Como se explicó en el apartado 2.5.1, debido a que estas FPGA son volátiles, si queremos que se guarde un circuito de manera permanente en la tarjeta, lo guardaremos en la memoria *flash* (no volátil). Si al encender la tarjeta el *jumper* está en la posición *ROM*, la FPGA se programará con el circuito que se haya grabado en dicha memoria *flash*. Como nosotros vamos a probar circuitos de manera continua, y no queremos grabar de manera permanente ningún circuito, lo ponemos en la otra posición.

A continuación se explica cómo se programa la tarjeta *Basys* con la versión 2 del *Adept*. Ejecuta el programa *Adept*: *Inicio* → *Todos los Programas* → *Digilent* → *Adept*. En la figura 2.20 se muestra la pantalla inicial cuando no hay ningún dispositivo conectado. Esto se puede ver en la parte derecha de la ventana.

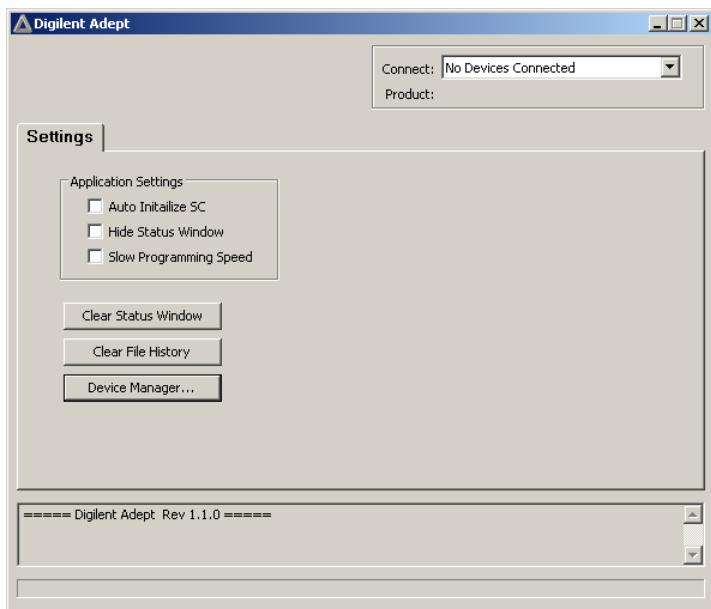


Figura 2.20: Pantalla inicial del Adept, sin dispositivos conectados

Si conectamos la tarjeta *Basys* al ordenador mediante el cable USB y el interruptor está en la posición *VUSB*, transcurrido un tiempo aparecerá que está conectada una tarjeta, en la parte derecha pondrá *Onboard USB* (ver figura 2.21). Si no saliese nada, prueba a pinchar en el menú que pone *No Devices Connected* a ver si cambia. Si no cambiase, prueba a cerrar el *Adept* y volverlo a arrancar. La primera vez que conectemos la tarjeta puede ser que Windows haya detectado un nuevo hardware y tengamos que dar permiso para la instalación de los *drivers* que vienen con la tarjeta (no tenemos que introducir un disco, la propia le proporciona tiene los *drivers*).

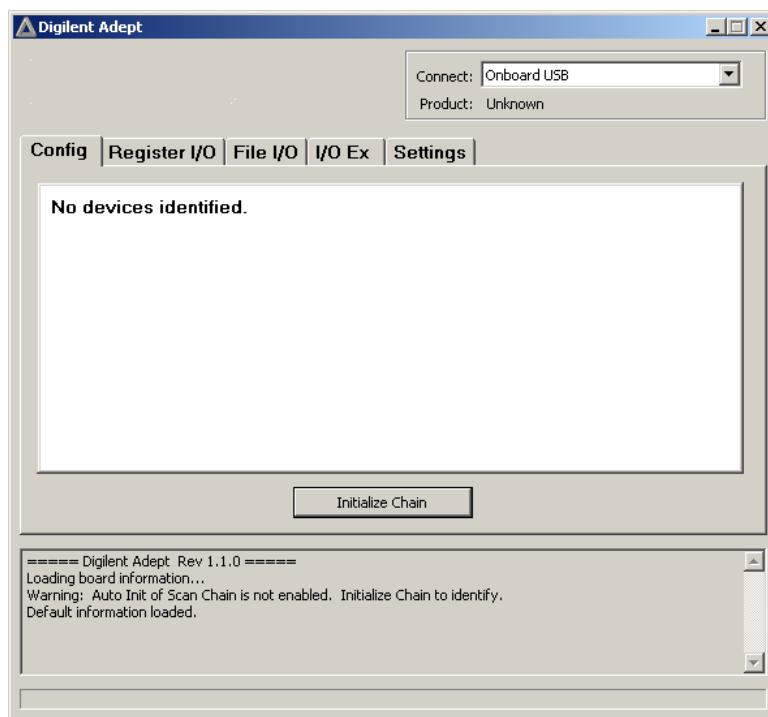


Figura 2.21: Pantalla del Adept que ha detectado algún dispositivo conectado

Ahora pinchamos en *Initialize Chain* y, como se ve en la figura 2.22, aparecerán dos componentes: uno es la FPGA y otro la memoria (PROM). Indicará también el modelo, que en nuestro caso la FPGA es la xc3s100e.

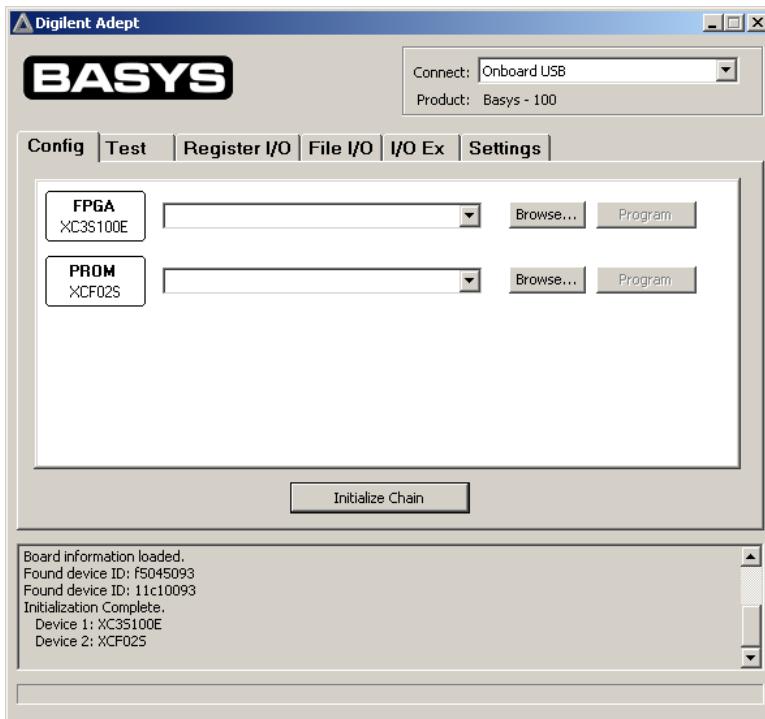


Figura 2.22: Pantalla del Adept que ha detectado la cadena JTAG de dispositivos (FPGA y PROM)

A continuación pinchamos en el botón *browse* correspondiente a la FPGA, y buscamos el fichero `led0.bit` que hemos generado en nuestro proyecto. Aparecerá un *warning* indicando el mensaje : "Startup clock for this file is 'CCLK' instead of 'JTAG CLK'. Problems will likely occur. Associate config file with device anyway?". A este mensaje contestamos **Sí**.

Otra cosa es que salga un mensaje de error indicando: "Unable to associate file with device due to IDCODE conflict". Esto es un error que indica que la FPGA que hemos escogido no es la misma que la que está en la placa. En este caso, tendremos que verificar que el modelo de FPGA sea el mismo. Si no fuese así, lo tendremos que cambiar. Consulta el apartado 2.6 si no sabes cómo se hace.

Si todo ha salido bien, sólo tenemos que pinchar en *Program* y se programará la FPGA. El resultado será que se encenderá el LED `LD0`, y también se encenderá otro LED que indica que se ha programado la FPGA (`LD-D`).

## 2.6. Cambiar el tipo de FPGA de un proyecto

Si en un proyecto queremos modificar el tipo de FPGA, ya sea porque vamos a cambiar de placa o porque nos hemos equivocado al principio poniendo las características (figura 2.7), en cualquier momento podemos realizar el cambio. Para ello, tenemos que seleccionar la FPGA en la subventana *Sources*, y pinchar con el botón izquierdo del ratón. En el menú desplegable que aparece tendremos que pinchar en *Properties* (figura 2.23). Tras esta operación aparecerá la ventana de la figura 2.7 que nos permitirá cambiar el modelo y las características de la FPGA. Ten en cuenta que según el cambio que hagamos, quizás también tengamos que cambiar el fichero `.ucf`, pues en él se indican los pines, y

éstos pueden ser distintos con el modelo de la FPGA (incluso pueden haber cambiado de nombre).

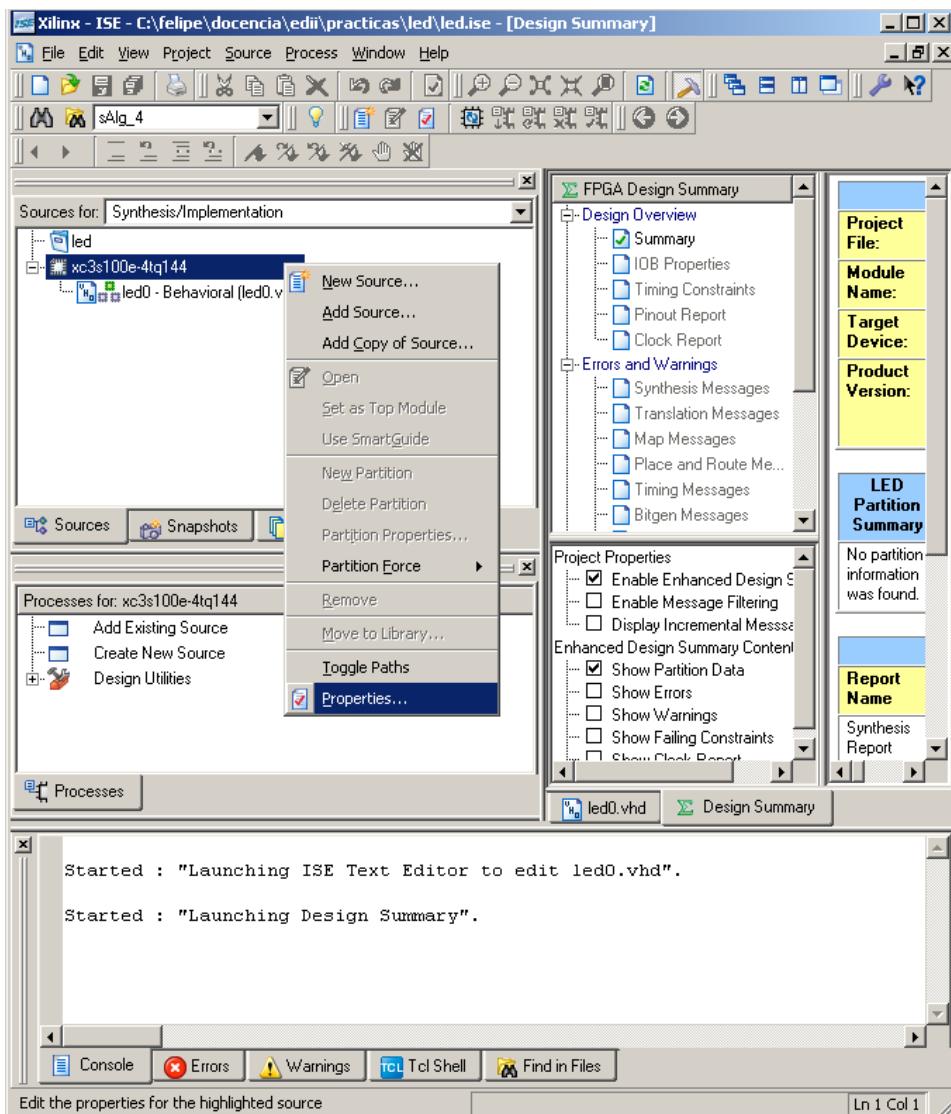


Figura 2.23: Procedimiento para cambiar las características de la FPGA

## 2.7. Trabajar desde varios ordenadores con un mismo proyecto

Cuando estamos trabajando en un ordenador y queremos seguir trabajando en otro, tenemos que hacer una copia de nuestro trabajo. Por ejemplo, esto es necesario cuando trabajas en los ordenadores de la universidad, si no haces copia de tu proyecto puede pasar que el próximo día alguien haya borrado todo tu trabajo. Esto incluso es importante aunque trabajes siempre en el mismo ordenador, ya que es importante que hagas copias de seguridad y que tengas un control de versiones de los diseños que van funcionando.

Cuando quieras hacer una copia de tu trabajo no tienes que copiar todo el directorio del proyecto. El directorio puede ocupar varios megabytes y realmente no es muy útil pasarlo de un ordenador a otro, pues muchas veces al intentarlo abrir el *ISE* da problemas. Lo mejor es que sólo copiemos los ficheros con extensión .vhd y .ucf. Estos ficheros son de texto y ocupan muy poco. Si has llegado a implementar el circuito, puedes

copiar el fichero `.bit`. Que es el fichero que se usa para programar la FPGA, así no tienes que volverlo a generar.

Estos ficheros `.vhd` y `.ucf` son los que vamos a copiar en nuestra memoria USB o nos lo enviamos por correo electrónico.

Cuando queramos importar nuestro proyecto a un nuevo ordenador, lo que tenemos que hacer es:

- Crear un nuevo proyecto: *File → New Project* (recuerda el apartado 2.3)
- Incluimos el nombre y el camino (*path*) del proyecto (recuerda la figura 2.4)
- Seleccionamos las características de la FPGA (figura 2.7)
- En la ventana: *New Project Wizard - Create New Source*, pinchamos en *Next*
- En la ventana: *New Project Wizard - Add Existing Sources*, pinchamos en *Add*, y buscamos todos los ficheros `.vhd` y `.ucf`. Nos aseguramos que está marcado la casilla *Copy to Project* y pinchamos en *Next* y luego en *Finish*.

Con esto ya podemos seguir trabajando con nuestro proyecto. Si posteriormente quisiéramos añadir nuevos ficheros `.vhd` al proyecto sólo tendríamos que pinchar en *Source → Add Source* y buscar el fichero que queremos añadir.

## 2.8. Conclusión

Con esto hemos terminado el primer ejercicio. Hemos repasado conceptos del año pasado. Hemos comprobado que la placa y la herramienta funcionan, y ya podemos probar otros diseños más complejos.

Resumiendo, los conceptos principales de esta práctica:

- Los comentarios en VHDL empiezan con dos guiones: `--`
- El VHDL no distingue entre mayúsculas y minúsculas
- Para las señales binarias normalmente se utiliza el tipo `STD_LOGIC`
- Un diseño VHDL consta como mínimo de una entidad y una arquitectura
- La entidad define las entradas y salidas del diseño
- La arquitectura define el diseño internamente
- Normalmente es necesario hacer referencia a la biblioteca del IEEE: `STD_LOGIC_1164`.
- Es importante hacer copias de seguridad de nuestro trabajo. Para esto, basta con copiar los ficheros con extensión `.vhd` y `.ucf`. Añadir estos ficheros a un proyecto es una tarea sencilla.

### 3. Sentencias concurrentes

El diseño anterior era muy básico, simplemente encendía un LED y no teníamos posibilidad de interactuar con el circuito, por ejemplo, mediante los pulsadores e interruptores. Ahora, en esta práctica tendremos la posibilidad de crear funciones lógicas y bloques que estudiamos el año pasado.

A partir de ahora no se va a realizar explicaciones tan detallada como la del anterior ejercicio, si durante la ejecución de este ejercicio has olvidado algún paso, repasa lo explicado en el ejercicio anterior.

Analizando las conexiones de los interruptores de la *Pegasus* (figura 3.1) vemos que cuando el interruptor está hacia arriba transmite un 1-lógico a la entrada de la FPGA. En la placa, a la derecha del interruptor sw0 se indica que hacia arriba son 3,3V y hacia abajo son 0V.

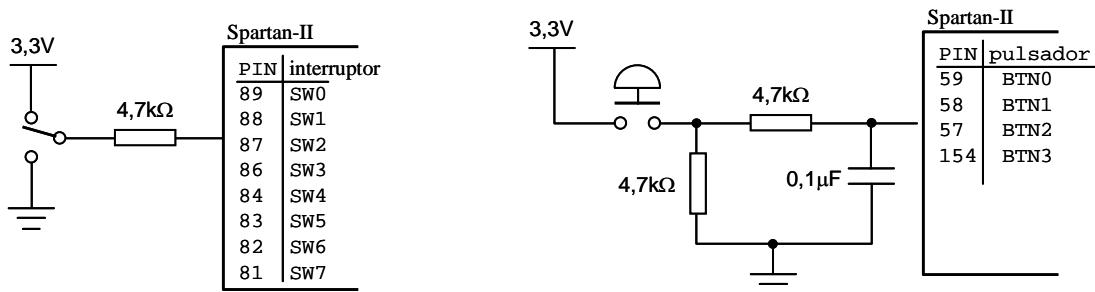


Figura 3.1: Esquema eléctrico de las conexiones de los interruptores y pulsadores en la placa Pegasus

Para los pulsadores, cuando están pulsados transmiten un 1-lógico a la entrada y cuando no están pulsados ponen 0 voltios a la entrada. Observa que los pulsadores tienen un pequeño circuito con resistencias y un condensador, esto es para eliminar los rebotes que se producen al pulsar y así enviar una señal sin ruido. Este circuito que elimina rebotes no lo tiene la *Basys*, y por lo tanto, la señal de entrada de los pulsadores no es tan limpia, y puede parecer que se ha pulsado varias veces (rebote). El esquema eléctrico se muestra en la figura 3.2, en donde se puede observar que los pulsadores no tienen el condensador que sí tiene la tarjeta *Pegasus*.

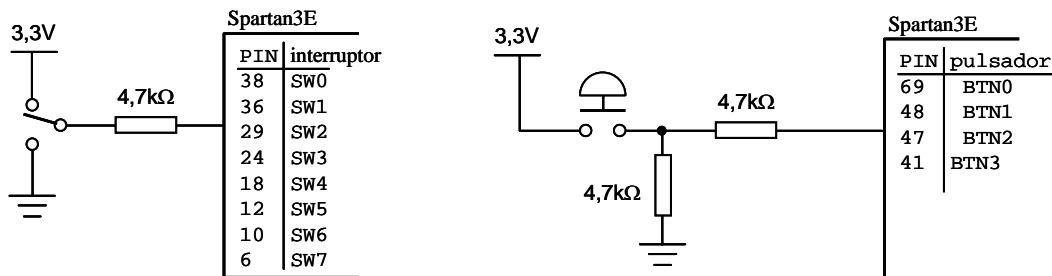


Figura 3.2: Esquema eléctrico de las conexiones de los interruptores y pulsadores en la placa Basys

En la práctica de hoy utilizaremos los interruptores y pulsadores como entradas de los circuitos que realicemos.

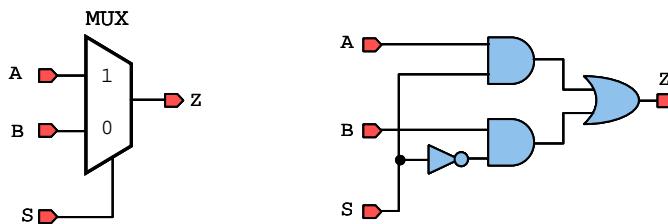
Antes de empezar con la práctica, se muestran a continuación las palabras reservadas del VHDL (figura 3.3). Estas palabras no se pueden usar para nombrar elementos de nuestro circuito: señales, entidades, arquitecturas, ....

abs	disconnect	is	out	sli
access	downto	label	package	sra
after	else	library	port	srl
alias	elsif	linkage	postponed	subtype
all	end	literal	procedure	then
and	entity	loop	process	to
architecture	exit	map	pure	transport
array	file	mod	range	type
assert	for	nand	record	unaffected
attribute	function	new	register	units
begin	generate	next	reject	until
block	generic	nor	return	use
body	group	not	rol	variable
buffer	guarded	null	ror	wait
bus	if	of	select	when
case	impure	on	severity	while
component	in	open	signal	with
configuration	inertial	or	shared	xnor
constant	inout	others	sla	xor

Figura 3.3: Lista de palabras reservadas del VHDL

### 3.1. Diseño de un multiplexor

El primer ejercicio consistirá en crear el diseño de un multiplexor de dos entradas de datos de un bit. Por tanto, este multiplexor tendrá una señal de selección (un bit).



Si  $S=1$ , por la salida Z tendremos A  
Si  $S=0$ , por la salida Z tendremos B

Figura 3.4: Esquema del multiplexor (izquierda) y diseño en puertas (derecha)

#### 3.1.1. Diseño usando puertas lógicas

Si recordamos del año pasado, un multiplexor en puertas lógicas se describe como se muestra en la derecha de la figura 3.4.

Así que creamos un nuevo proyecto llamado `gate_mux` (recuerda ponerlo en: `C:\practicas\ed2\tunombre`). Crea un nuevo módulo VHDL llamado también `GATE_MUX` con los puertos llamados igual que los de la figura 3.4. La arquitectura, en vez de llamarla BEHAVIORAL llámala `GATE`, para indicar que está en el nivel de puertas.

Dentro de la arquitectura, incluye la sentencia concurrente que define el esquemático en el nivel de puertas del multiplexor (derecha en la figura 3.4). Esta sentencia se muestra en el código 3-1.

```
Z <= (A and S) or (B and (not S));
```

*Código 3-1: Sentencia concurrente que define el multiplexor en puertas*

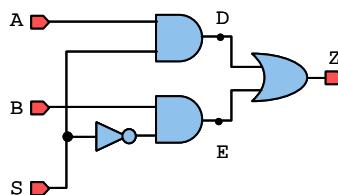
A partir del código 3-1 podemos observar que para un diseño en el nivel de puertas, la sintaxis es muy similar a la sintaxis utilizada en el álgebra de Boole (que estudiamos el año pasado). El uso de paréntesis es muy importante para determinar que operador booleano se aplica en cada caso.

Prueba a implementar el circuito en la placa haciendo que el puerto **A** sea el interruptor **SW0**, el puerto **B** sea el interruptor **SW1**, y que la señal de selección sea el pulsador **BTN0**. Por último, asocia la salida **z** con el LED **LDO**.

Una vez implementado, comprueba que funciona correctamente, esto es, que según esté el pulsador presionado, la salida toma el valor del primer interruptor o del segundo.

### 3.1.2. Diseño usando varias sentencias concurrentes

El código 3-1 es perfectamente válido. Sin embargo, cuando el número de puertas lógicas es elevado, nos puede interesar dividir la sentencia de asignación en varias sentencias más sencillas. Esto se hace creando señales intermedias. La figura 3.5 muestra el diseño en puertas del multiplexor incluyendo las señales intermedias **D** y **E**.

*Figura 3.5: Esquema en puertas del multiplexor con señales intermedias*

Antes de ser usadas, estas señales intermedias deben ser primero declaradas en la arquitectura. Las señales se declaran antes del **begin** de la arquitectura. Por tanto, nuestro diseño queda como muestra el código 3-2:

```

architecture GATE of GATE_MUX is
    signal D, E : STD_LOGIC;
begin
    D <= A and S;
    E <= (not S) and B;
    Z <= D or E;
end GATE;

```

*Código 3-2: Sentencias concurrentes que definen el multiplexor en puertas (equivalente al código 3-1)*

Ahora hemos usado tres sentencias concurrentes que son equivalentes a la sentencia del código 3-1. Puedes probar, siquieres, que funciona igual.

Una característica fundamental de las sentencias concurrentes es que, como su nombre indica, son **concurrentes**. Esto es, que se ejecutan todas a la vez. Que es como funcionan los circuitos (el hardware): todo está ejecutándose simultáneamente. Por lo tanto, **da igual el orden en que se escriban**, pues no hay unas que vayan antes o después. Así, las sentencias del código 3-2 se podrían poner en cualquier otro orden. Por ejemplo, el código 3-3 es totalmente equivalente al del código 3-2.

```

architecture GATE of GATE_MUX is
  signal D, E : STD_LOGIC;
begin
  Z <= D or E;
  D <= A and S;
  E <= (not S) and B;
end GATE;

```

*Código 3-3: Otra versión de las sentencias concurrentes que definen el multiplexor en puertas (equivalente a los códigos 3-1 y 3-2)*

Si quieras, puedes implementar en la placa el código 3-3 para comprobar que funciona igual.

### 3.1.3. Uso de sentencias condicionales

Usar el VHDL solamente con expresiones booleanas nos limita las capacidad de diseño. Ahora vamos a realizar el mismo diseño usando sentencias condicionales, lo que a los humanos nos resultará mucho más entendible que las expresiones booleanas.

Podemos utilizar una sentencia concurrente condicionada como la mostrada en el código 3-4. Que es bastante más entendible que el código en el nivel de puertas. Ya que como dice el código, a z se le asigna a cuando s='1' y si no, se le asigna b.

```
Z <= A when S='1' else B;
```

*Código 3-4: Sentencia concurrente condicionada que define el multiplexor*

Prueba a implementar este diseño y comprueba que funciona igual.

### 3.1.4. Uso de procesos

Otra alternativa es usar procesos, que los veremos con más detalle más adelante. Dentro del proceso, las sentencias se tratan secuencialmente y no de manera concurrente como vimos que ocurre con las sentencias concurrentes puestas directamente en la arquitectura. Así que **el orden de las sentencias dentro de un proceso sí es importante**.

El proceso empieza con su nombre, a lo que le siguen dos puntos y la palabra reservada PROCESS. Después de ésta se pone la **lista de sensibilidad**, donde se deben poner todas las señales que se leen en el proceso. Las señales que se leen son: las señales que están dentro de las condiciones (por ejemplo dentro de la sentencia if) y las señales que están en la parte derecha de las asignaciones.

El proceso que implementa nuestro multiplexor se muestra en el código 3-5.

```

P_MUX: Process (A, B, S)    -- lista de sensibilidad: A, B, S
begin
  if S='1' then      -- S se lee
    Z <= A;          -- A se lee
  else
    Z <= B;          -- B se lee
  end if;
end process;

```

*Código 3-5: Sentencia concurrente condicionada que define el multiplexor*

El VHDL es un lenguaje muy amplio, y hay otras maneras con las que se puede describir el multiplexor. Sin embargo, por ahora no las veremos y pasaremos a describir otros diseños.

### 3.2. Diseño de un multiplexor de 4 alternativas

El multiplexor anterior era demasiado sencillo. Ahora queremos realizar un multiplexor con cuatro alternativas. La figura 3.6 muestra el esquema de este multiplexor.

El año pasado vimos cómo se realizaba en puertas este multiplexor y también cómo se realizaba con multiplexores de dos alternativas. Ahora simplemente veremos cómo se describe en VHDL.

Para poder seleccionar 4 alternativas la señal de selección (*s*) debe de ser de 2 bits. Por tanto necesitamos definir un vector. Para esto se utiliza el tipo STD\_LOGIC\_VECTOR.

Para realizar el diseño, creamos un nuevo proyecto llamado *mux4alt* y al definir los puertos debemos crear la entrada *s* como un vector. Para ello, al definir los puertos, activamos el campo *Bus* del puerto *s*, e indicamos que el *MSB* (bit más significativo<sup>11</sup>) es el bit número 1, y el *LSB* (bit menos significativo<sup>12</sup>) es el 0. Por tanto, tendrá dos bits de ancho de bus. Observa la figura 3.7 para saber cómo debes crearlo.

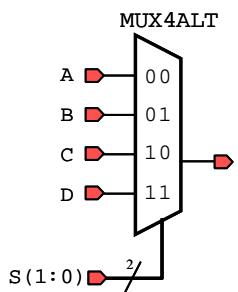


Figura 3.6: Multiplexor de 4 alternativas

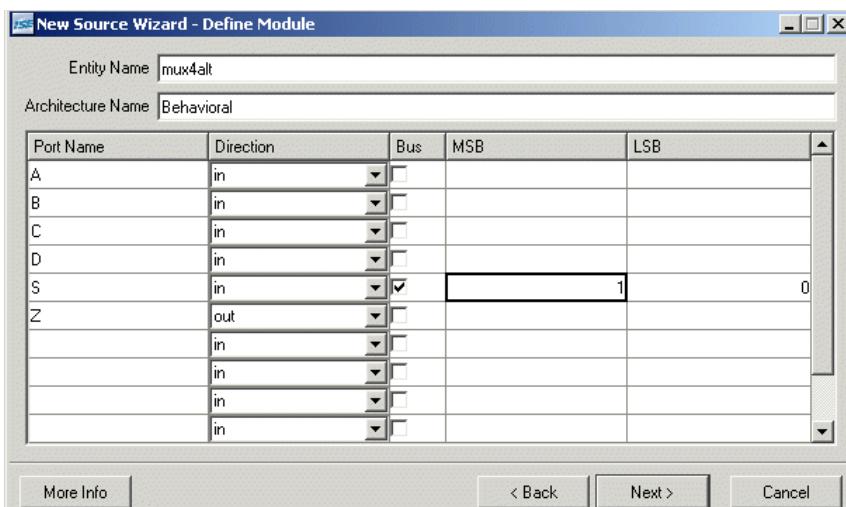


Figura 3.7: Definición de los puertos de entrada y su ancho

El resultado es la entidad mostrada en el código 3-6.

```

entity mux4alt is
  Port ( A : in  STD_LOGIC;
         B : in  STD_LOGIC;
         C : in  STD_LOGIC;
         D : in  STD_LOGIC;
         S : in  STD_LOGIC_VECTOR (1 downto 0);
         Z : out  STD_LOGIC);
end mux4alt;
  
```

Código 3-6:Entidad del multiplexor de 4 alternativas

El tipo STD\_LOGIC\_VECTOR es un vector de STD\_LOGIC y normalmente se definen con rango descendente terminado en cero. El rango descendente se especifica con la palabra *downto*.

<sup>11</sup> MSB: del inglés *Most Significant Bit*

<sup>12</sup> LSB: del inglés *Least Significant Bit*

La sentencia concurrente que describe este multiplexor se muestra en el código 3-7. Esta sentencia es una extensión para 4 alternativas del código 3-4. Fíjate que la comparación del valor de la señal s se hace con comillas dobles por ser un vector.

```
Z <= A when S="00" else
B when S="01" else
C when S="10" else
D;
```

Código 3-7: Sentencia concurrente condicionada que define el multiplexor de cuatro alternativas

Si lo quisiésemos realizar con un proceso tenemos dos maneras de hacerlo, mediante `if` (código 3-8) o mediante `case` (código 3-9). Es muy importante que la última alternativa siempre cierre todas las posibilidades. En el caso del `IF` se debe terminar con un `ELSE`. Y en el caso del `CASE` se debe de terminar con un "when others", que significa "para el resto de posibilidades". Se deben cerrar todas las posibilidades porque, en otro caso, la señal asignada no recibiría valor y por lo tanto, se crearía un elemento de memoria (un *latch* probablemente) para guardar el valor que tenía anteriormente. Esto ya lo veremos en el capítulo 5, cuando trabajemos con los elementos de memoria.

```
P_MUX: Process (A,B,C,D,S)
begin
  if S="00" then
    Z <= A;
  elsif S="01" then
    Z <= B;
  elsif S="10" then
    Z <= C;
  else
    Z <= D;
  end if;
end process;
```

Código 3-8: Proceso con sentencia IF

```
P_MUX: Process (A,B,C,D,S)
begin
  case S is
    when "00" =>
      Z <= A;
    when "01" =>
      Z <= B;
    when "10" =>
      Z <= C;
    when others =>
      Z <= D;
  end case;
end process;
```

Código 3-9: Proceso con sentencia CASE

Ahora implementa en la FPGA este multiplexor de 4 alternativas empleando las tres formas que hemos visto (código 3-7, 3-8 y 3-9). Asocia la señal s a los dos primeros pulsadores, las cuatro alternativas (`A`, `B`, `C`, `D`) a los cuatro primeros interruptores, y la salida z al primer LED (`LED0`).

### 3.3. Diseño de un multiplexor de 4 bits de dato y dos alternativas

Ahora haremos un multiplexor de dos alternativas, pero cada alternativa tendrá cuatro bits. Por tanto, la salida también tendrá 4 bits.

El esquema de este multiplexor se muestra en la figura 3.8

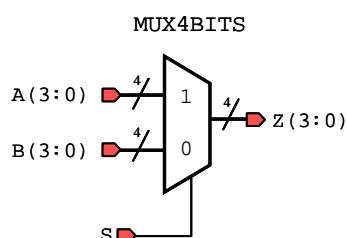


Figura 3.8: Multiplexor de 4 bits de dato

La entidad será como la mostrada en el código 3-10.

```
entity mux4bits is
  Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
         B : in STD_LOGIC_VECTOR (3 downto 0);
         S : in STD_LOGIC;
         Z : out STD_LOGIC_VECTOR (3 downto 0));
end mux4bits;
```

Código 3-10: Entidad del multiplexor de 4 alternativas

La arquitectura será igual que la usada para el multiplexor de dos alternativas con datos de un bit, ya que la asignación de señales utiliza la misma expresión independientemente de que las señales sean vectores o bits. Lo que sí se debe cumplir es que la señal asignada tenga el mismo número de bits que la asigna el valor.

Por tanto, se podrá utilizar la misma sentencia que se utilizó en el código 3-4 o el proceso del código 3-5. Eso sí, sabiendo que ahora las señales `A`, `B` y `z` son de 4 bits y no de un bit.

Implementa este multiplexor asociando el puerto `A` a los cuatro primeros interruptores (`sw3`, `sw2`, `sw1` y `sw0`), el puerto `B` al resto de interruptores (`sw7`, `sw6`, `sw5` y `sw4`). La señal de selección al primer pulsador (`BTN0`), y la salida `z` a los cuatro primeros LED.

Se deja como ejercicio implementar un multiplexor de cuatro alternativas, con datos de cuatro bits.

---

### 3.4. Conclusiones

Resumiendo, los conceptos principales de esta práctica:

- Existe un grupo de palabras reservadas en VHDL que no se pueden utilizar para nombrar elementos del circuito: señales, entidades, arquitecturas, componentes, ...
- Las sentencias concurrentes, dentro de la arquitectura, se ejecutan a la vez. Por lo tanto, no importa el orden en que aparezcan.
- Dentro de los procesos, el orden de las sentencias sí importa. Por lo tanto, se tratan secuencialmente.
- El VHDL es un lenguaje muy amplio y existen muchas maneras de describir la misma cosa
- En VHDL los valores vectoriales van en comillas dobles, por ejemplo `S="00"`.
- Al implementar un multiplexor es importante que la última condición sea una condición por defecto que cubra el resto de los casos, como lo son "`when others`" o "`else`". Con esto evitamos generar un elemento de memoria (se verá en el capítulo 5)

## 4. Codificadores, decodificadores, convertidores de código

En esta práctica vamos a complicar diseños de las prácticas anteriores

### 4.1. Convertidor de binario a 7 segmentos

En este ejercicio vamos a modificar el diseño del multiplexor de 4 bits de datos (sección 3.3) para que además de mostrar el dato seleccionado por los cuatro LED, muestre el dato por un *display* de siete segmentos.

El esquema del circuito se muestra en la figura 4.1. En ella se puede observar el multiplexor que hicimos en el apartado 3.3. La salida de este multiplexor la queremos convertir a siete segmentos. En la derecha de la figura se han incluido los puertos. Además de los que aparecen en el esquema, se han añadido los ánodos de los *displays* (puerto AN). Quizás recuerdes del año pasado que con los ánodos se controla qué *displays* van a lucir. Tanto los ánodos como los segmentos de los *displays* son activos a nivel bajo. Esto es, para que luzca un segmento hay que poner un cero-lógico. Y para activar un *display* tengo que poner un cero-lógico en el ánodo correspondiente.

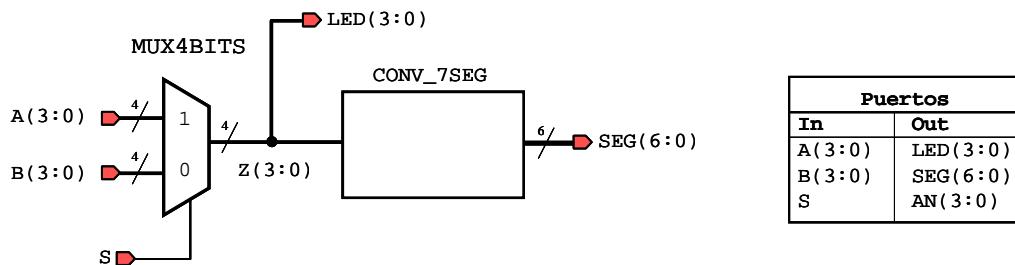


Figura 4.1: Multiplexor y salida a 7 segmentos

Así que crea un nuevo proyecto llamado `mux7seg`, y dentro de él crea un nuevo módulo VHDL que tenga los puertos mostrados en la tabla de la derecha de la figura 4.1.

Dentro de la arquitectura deberás declarar la señal `z`, que la asignarás al puerto de salida `LED` y que usarás para convertirla en los siete segmentos. No podrás usar `z` como puerto: esto es un concepto importante en VHDL ya que **los puertos de salida no se pueden leer**. Por tanto, es necesario usar una variable intermedia `z` distinta del puerto de salida `LED`. Como la señal `z` la vamos a utilizar en el convertidor de binario a siete segmentos, la señal `z` no podrá ser un puerto de salida al contrario de como lo hicimos en el ejercicio del apartado 3.3.

Para hacer el convertidor de binario a siete segmentos tenemos que recordar la visualización de cada número. En la figura 4.2 se muestran los 16 números que podemos representar con 4 bits (de 0 a 15).

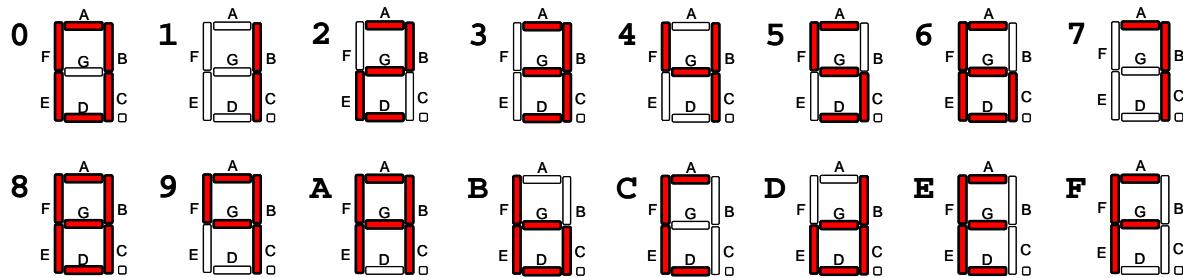
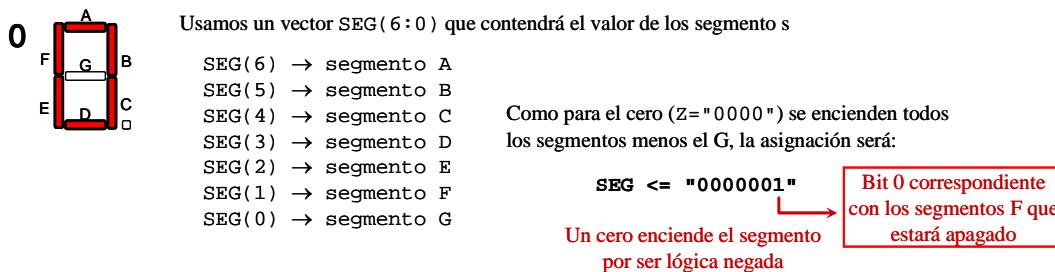


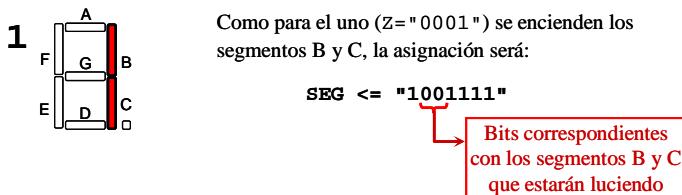
Figura 4.2: Los 16 números representables con 4 bits

Así que para realizar el convertidor tenemos que examinar qué número tenemos que mostrar y encender los segmentos correspondientes a dicho número. En la figura 4.3 se muestran los ejemplos para tres casos: los números 0, 1 y 2. Como se puede ver, arbitrariamente se ha asignado el bit más significativo de SEG al segmento A y el menos significativo (el bit 0) al segmento G. Otro orden se podía haber escogido, pero lo importante es que se correspondan adecuadamente a la asignación de pines de la FPGA. En la figura se han mostrado tres ejemplos, ahora tienes tú que deducir el resto (de manera parecida a como lo hiciste en la práctica del año pasado).

Si  $Z = "0000"$  → mostramos el cero



Si  $Z = "0001"$  → mostramos el uno



Si  $Z = "0010"$  → mostramos el dos

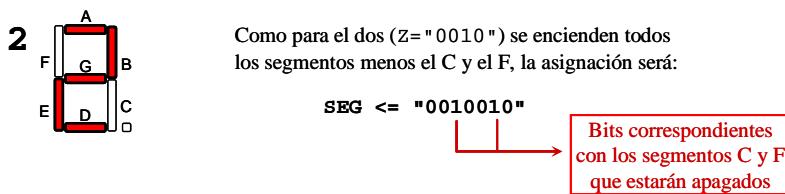


Figura 4.3: Valores del vector de los segmentos (SEG) para los números 0, 1 y 2

Las sentencias concurrentes VHDL de la arquitectura deben tener el aspecto de la figura 4.4. Se puede apreciar que hay cuatro sentencias concurrentes. Como ya hemos dicho, por la concurrencia del hardware, el orden de estas sentencias es indiferentes. La última sentencia, correspondiente al convertidor a siete segmentos, se ha dejado incompleta para que la termines tú. Puedes observar que en la sentencia del convertidor se han incluido comentarios para facilitar la comprensión del código. Esto es muy importante para

facilitar la comprensión del código, tanto para otras personas que lo lean, o para ti mismo cuando lo vuelvas a ver dentro de unas semanas y no te acuerdes de nada de lo que hiciste en su día.

Fíjate que se han incluido las dos últimas alternativas. En la última no se ha incluido la condición (el `when` se ha comentado), por lo tanto, en cualquier otro caso, se asigna el último valor<sup>13</sup>

```

Z <= A when S='1' else B; } multiplexor

LED <= Z; } Asignación de las señal Z a los LED
            (no se puede usar Z como puerto de salida)

AN <= "1110"; } Encendemos el ánodo 0

SEG <= "0000001" when Z=="0000" else -- 0: A,B,C,D,E,F
      "1001111" when Z=="0001" else -- 1: B,C
      "0010010" when Z=="0010" else -- 2: A,B,D,E,G
      "0000110" when Z=="0011" else -- 3: A,B,C,D,G
      ....
      .... continua tú con el resto de casos
      ....
      "0110000" when Z=="1110" else -- E(14): A,D,E,F,G
      "0111000"; -- when Z=="1111"; -- F(15): A,E,F,G

            { comentarios para ayudar a identificar
              el número y los segmentos que lucen

            { Primeros 4 números
              del convertidor a
              siete segmentos

            { Dos últimos números
              del convertidor

En la última alternativa no se pone
condición para evitar generar un latch

```

Figura 4.4: Sentencias concurrentes de la arquitectura

Ahora, termina de hacer el convertidor a siete segmentos e implementa el diseño en la FPGA. Ten especial cuidado de asignar los pines adecuadamente. Puede ser que te salgan al revés por haber asignado los pines en sentido contrario, esto es, el segmento A donde iría el segmento G, el B donde el F ...

Cuando lo hayas hecho, comprueba que todos los números se muestran correctamente.

Date cuenta lo sencillo que resulta hacer el convertidor con VHDL frente a hacerlo con esquemáticos. Ahora es el sintetizador (*ISE-Webpack*) quien se encarga de optimizar el circuito, así que no tenemos que preocuparnos de hacer los mapas de Karnaugh.

## 4.2. Decodificador de 2 a 4

Ahora haremos un decodificador de 2 a 4. Este ejercicio lo hicimos el año pasado con esquemáticos. El bloque del circuito y su tabla de verdad se muestra en la figura 4.5. Recuerda que un decodificador activa la salida correspondiente al número codificado en las entradas. Por ejemplo, si en la entrada hay un 2 codificado en binario, se activa la salida número 2. Sólo una salida puede estar activa en un decodificador. Además, el decodificador tiene una señal de habilitación `E` que hace que no se active ninguna señal de salida en caso de que valga cero (suponiendo que la señal `E` se activa a nivel alto).

<sup>13</sup> Aunque no hay ningún caso más, es mejor poner la condición por defecto. Tanto por si se nos ha olvidado incluir alguna otra, como por si el sintetizador no se da cuenta de que no hay más condiciones, y en tales casos, nos generaría un elemento de memoria (en estos casos un *latch*).

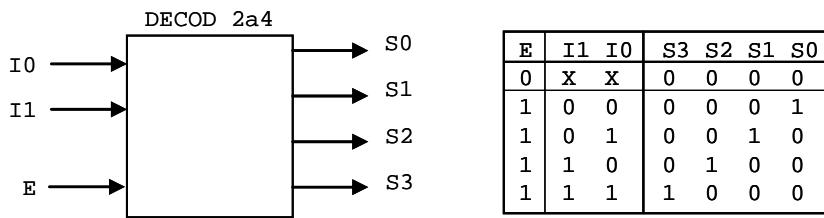


Figura 4.5: Puertos del decodificador y tabla de verdad

El decodificador se hace de manera similar al convertidor de siete segmentos del ejercicio anterior, sólo que cambian las salidas que se activan según el número que se tenga en la entrada. La mayor novedad es la señal de habilitación E.

Lo que vamos a hacer en este diseño es mostrar en el display el número codificado en los cuatro primeros interruptores (sw<sub>3</sub>, sw<sub>2</sub>, sw<sub>1</sub>, sw<sub>0</sub>). El número también se mostrará en los cuatro primeros LED (de LD<sub>3</sub> a LD<sub>0</sub>). Sin embargo, en vez de mostrarlo en el primer *display*, los interruptores sw<sub>7</sub> y sw<sub>6</sub> indicarán el *display* por el que se mostrará. Para ello necesitamos un decodificador, que decodifique el número representado por estos interruptores active el ánodo correspondiente. El esquema del circuito que queremos realizar se muestra en la figura 4.6.

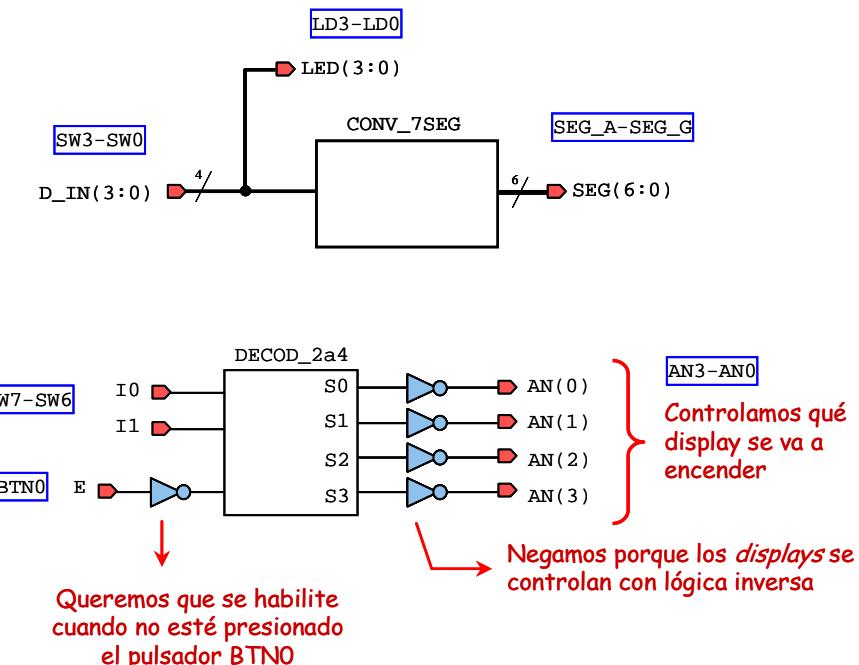


Figura 4.6: Esquema del circuito del decodificador completo

En el dibujo podemos ver que el circuito tiene dos partes diferenciadas. La parte de arriba (en la figura) realiza la conversión de siete segmentos, esta parte ya la sabes hacer del apartado anterior. La parte de abajo se encarga de controlar qué *display* va a lucir. Cuando I="00" se debe activar el primer *display*, cuando I="01" se debe activar el segundo *display*, el tercero cuando I="10" y el cuarto cuando I="11". Todo esto siempre cuando no esté presionado el pulsador BTN0, ya que va a funcionar como habilitación.

El decodificador con habilitación lo podemos hacer con sentencias concurrentes (código 4-1) o con un proceso (código 4-2). Ambas formas producen circuitos equivalentes, incluso hay más formas de hacerlo.

En el decodificador realizado con la sentencia concurrente (código 4-1) se ha utilizado una señal auxiliar `S_DECOD` que luego se va a invertir debido a que los ánodos funcionan con lógica negada. En el código del proceso las salidas se han codificado directamente con lógica negada. Algo similar se ha hecho con la señal de habilitación, en el código 4-1 se ha negado y luego se ha comprobado si vale 0. Mientras que en el proceso directamente se comprueba si vale uno. El VHDL permite elegir la manera que más te guste. El código 4-1 usa inversores y se parece más al esquema de la figura 4.6. El código 4-2 quizás sea más fácil de entender.

```
E_NEG <= NOT E;

S_DECOD <= "0000" when E_NEG='0' else
              "0001" when I="00" else
              "0010" when I="01" else
              "0100" when I="10" else
              "1000"; -- when I="11"

-- invertimos para tener logica negada
AN <= NOT (S_DECOD);
```

Código 4-1:Decodificador con sentencia concurrente

```
P_DECOD: Process (E, I)
begin
  if E='1' then
    AN <= (others =>'1');
    -- AN <= "1111";
  else
    case I is
      when "00" =>
        AN <= "1110";
      when "01" =>
        AN <= "1101";
      when "10" =>
        AN <= "1011";
      when others => -- "11"
        AN <= "0111";
    end case;
  end if;
end process;
```

Código 4-2:Decodificador con proceso

Por último, fíjate en la primera asignación del código 4-2:

```
AN <= (others => '1');
```

Código 4-3:Una forma de asignar el mismo valor a todos los bits de un vector

Esta expresión asigna un uno a todos los bits de la señal `AN`. Con ello nos ahorraremos contar el número de bits que tenemos que asignar.

Ya es hora de implementar el circuito propuesto: Crea un nuevo proyecto llamado `decod2a4` y crea un nuevo módulo VHDL con los puertos mostrados en la figura 4.6. Los puertos que tengan más de un bit, créalos como vectores. En la figura 4.6 se indica a qué elementos de la placa debes asignar los puertos. Por ejemplo, la entrada `I(1:0)` la asocias a los interruptores `SW7` y `SW6`. La entrada `D_IN(3:0)` la asocias a los interruptores `SW3`, `SW2`, `SW1` y `SW0`. Y así todos los demás.

Una vez que lo tengas implementado comprueba:

- Que el *display* muestra correctamente el número codificado en los 4 primeros interruptores (`SW3-SW0`)
- Que al cambiar los interruptores `SW7` y `SW6` se cambia el *display* que luce.
- Que al presionar el pulsador de la habilitación (`BTN0`) ningún *display* luce
- Que el número codificado en los 4 primeros interruptores también se muestra por los 4 primeros LED.

### 4.3. Codificador de 8 a 3

Ahora vamos a realizar la operación inversa a la decodificación. Tendremos 8 entradas sin codificar (los 8 interruptores) y mostraremos en el primer *display* de siete segmentos el número de interruptor que está activo. Hay ciertas condiciones:

- En caso de que haya más de un interruptor activo, se mostrará el mayor (el más significativo es prioritario).
- En caso de que no haya ningún interruptor encendido, no mostraremos nada por el *display*.
- El número binario también se mostrará por los 3 primeros LED.
- El codificador tendrá una señal de habilitación *EI*, si ésta está deshabilitada se mostrará un signo menos (segmento G) en los cuatro *displays*.
- La señal de habilitación *EI* será el pulsador *BTN0*. Cuando esté pulsado, se deshabilitará el codificador.

Para recordar del año pasado, el esquema y la tabla de verdad del codificador se muestra en la figura 4.7. Un posible esquema del circuito se muestra en la figura 4.8, y en la figura 4.9 hay una descripción VHDL.

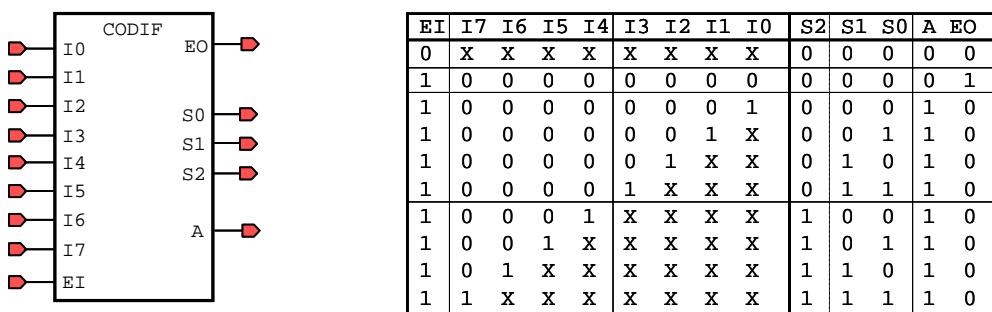


Figura 4.7: Esquema y tabla de verdad del codificador de 8 a 3

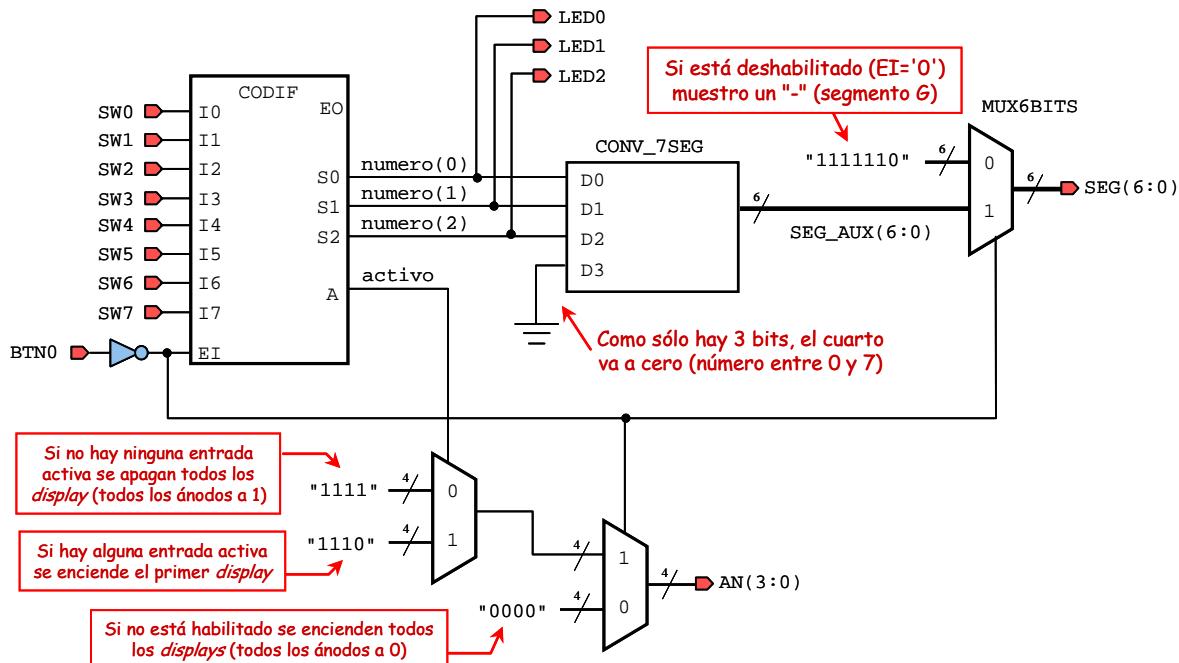


Figura 4.8: Esquema de bloques del codificador

```

P_CODIF: Process (BTN0, SW)
begin
    if BTN0 = '1' then
        numero <= "000";
        activo <= '0';
    else
        activo <= '1';
        if SW(7) = '1' then
            numero <= "111";
        elsif SW(6) = '1' then
            numero <= "110";
        elsif SW(5) = '1' then
            numero <= "101";
        elsif SW(4) = '1' then
            numero <= "100";
        elsif SW(3) = '1' then
            numero <= "011";
        elsif SW(2) = '1' then
            numero <= "010";
        elsif SW(1) = '1' then
            numero <= "001";
        elsif SW(0) = '1' then
            numero <= "000";
        else
            numero <= "000";
            activo <= '0';
        end if;
    end if;
end process;

LED <= numero; → Lleva el número codificado a los LED

```

Para no ponerla en todas las condiciones, se puede poner la asignación por defecto al principio

Codifica los interruptores en un número binario

Y cuando haya que asignar otro valor se incluye en esa condición

```

seg_aux <= "0001111" when numero="111" else -- 7
          "0100000" when numero="110" else -- 6
          "0100100" when numero="101" else -- 5
          "1001100" when numero="100" else -- 4
          "0000110" when numero="011" else -- 3
          "0010010" when numero="010" else -- 2
          "1001111" when numero="001" else -- 1
          "0000001"; -- when numero="000" -- 0

```

Convierte el número a siete segmentos

```

SEG <= seg_aux when BTN0='0' else "1111110"; → Inhabilita los segmentos si se pulsa BTN0 (pone un signo "-")

```

AN <= "0000" when BTN0='1' else
 "1111" when activo = '0' else
 "1110"; -- BTN0 = '1' AND activo = '1' } Controla qué displays lucen

Figura 4.9: Sentencias VHDL del esquema de la figura 4.8

No existe una única manera para describir el circuito. Por ejemplo, la sentencia que convierte el número a siete segmentos y la de habilitación de los segmentos se pueden agrupar en una sola. Con esto nos ahorramos la señal intermedia `seg_aux`. Compara los códigos 4-4 y 4-5, son equivalentes.

```

seg_aux <= "0001111" when numero="111" else
          "0100000" when numero="110" else
          "0100100" when numero="101" else
          "1001100" when numero="100" else
          "0000110" when numero="011" else
          "0010010" when numero="010" else
          "1001111" when numero="001" else
          "0000001"; -- when numero="000"

SEG <= seg_aux when BTN0='0' else "1111110";

```

Código 4-4: Habilitación con señal auxiliar

```

SEG <= "1111110" when BTN0='1' else
  "0001111" when numero="111" else
  "0100000" when numero="110" else
  "0100100" when numero="101" else
  "1001100" when numero="100" else
  "0000110" when numero="011" else
  "0010010" when numero="010" else
  "1001111" when numero="001" else
  "0000001"; -- when numero="000"

```

*Código 4-5: Habilitación dentro de la sentencia*

Otra alternativa es usar un sólo proceso para todo. Esta forma puede que sea más fácil de entender, aunque puede hacer que nos olvidemos del circuito que se va a generar (del esquema). Éste quizás es uno de los problemas más importantes de los lenguajes de descripción de hardware: distancian al diseñador del circuito que se va a generar, pudiendo hacerle creer que está haciendo un programa software.

```

LED  <= numero;

P_CODIF: Process (BTN0, SW)
begin
  if BTN0 = '1' then    -- esta deshabilitado
    AN <= "0000";      -- se encienden todos los displays
    numero <= "000";
    SEG <= "1111110";  -- mostramos un guion
  else
    AN <= "1110";      -- se enciende el display de la derecha
    if SW(7) = '1' then
      numero <= "111";
      SEG <= "0001111"; -- 7
    elsif SW(6) = '1' then
      numero <= "110";
      SEG <= "0100000"; -- 6
    elsif SW(5) = '1' then
      numero <= "101";
      SEG <= "0100100"; -- 5
    elsif SW(4) = '1' then
      numero <= "100";
      SEG <= "1001100"; -- 4
    elsif SW(3) = '1' then
      numero <= "011";
      SEG <= "0000110"; -- 3
    elsif SW(2) = '1' then
      numero <= "010";
      SEG <= "0010010"; -- 2
    elsif SW(1) = '1' then
      numero <= "001";
      SEG <= "1001111"; -- 1
    elsif SW(0) = '1' then
      numero <= "000";
      SEG <= "0000001"; -- 0
    else -- no hay ninguno activo
      numero <= "000";
      SEG <= "XXXXXXXX"; -- da igual, no se muestra ningun numero
      AN <= "1111"; --se apagan todos los displays
    end if;
  end if;
end process;

```

*Código 4-6: Otra alternativa al circuito de la figura 4.8, todo en un mismo proceso*

#### **4.4. Conclusiones**

Resumiendo, los conceptos principales de esta práctica:

- Los puertos de salida no se pueden leer, si lo quisiésemos leer tendríamos que crear una señal auxiliar que luego la asignaremos al puerto.
- Se pueden inicializar todos los bits de un vector a un mismo valor usando la expresión "`others => '1'`"

- El decodificador activa una única salida
- El VHDL y el sintetizador nos ahorra realizar las simplificaciones lógicas que hacíamos por Karnaugh
- Una señal sólo se debe escribir (asignar) en una sentencia concurrente o proceso. Si se asigna en más de una, estaremos creando un cortocircuito. Podríamos estar diciendo que un cable esté a 0 y 1 a la vez (en sintetizador *ISE* te dirá un error indicando *multisource*).
- Los lenguajes de descripción de hardware (HDL) junto con la síntesis automática nos facilitan mucho la tarea del diseño. Sin embargo, pueden hacer que nos perdamos la referencia sobre el circuito hardware que estamos diseñando, haciendo que diseñemos de forma similar a un programa software.

## 5. Elementos de memoria

Un elemento de memoria es un componente que es capaz de guardar (memorizar) un valor. Existen muchas maneras de modelar un elemento de memoria en VHDL, sin embargo, dentro del subconjunto de VHDL **sintetizable** sólo hay unas pocas maneras. El subconjunto de VHDL sintetizable es un estándar del VHDL [14] que restringe el lenguaje a un mínimo construcciones VHDL que los sintetizadores deben aceptar. Por tanto, no todo el VHDL es aceptado para sintetizarlo e implementarlo en la FPGA. Esto se debe a que el VHDL surgió para modelar y simular circuitos, y no para sintetizarlos. Por ello, el VHDL es mucho más amplio que lo que se puede sintetizar. En la práctica, cada sintetizador acepta un conjunto determinado para síntesis, y por ello se debe consultar el manual antes de empezar a usar un sintetizador nuevo [27].

A continuación, mediante ejemplos, veremos cómo se realizan elementos de memoria.

### 5.1. Biestable J-K

Empezaremos con uno de los ejemplos más sencillos que podemos probar: el biestable J-K. Para recordar del año pasado, en la figura 5.1 se muestra la tabla de verdad del biestable J-K. A la derecha se muestra el esquema del circuito que se quiere implementar, en donde se indican a qué elementos de la placa van conectados los puertos: Entradas a los pulsadores `BTN1` y `BTN0` y salida al LED `LD0`.

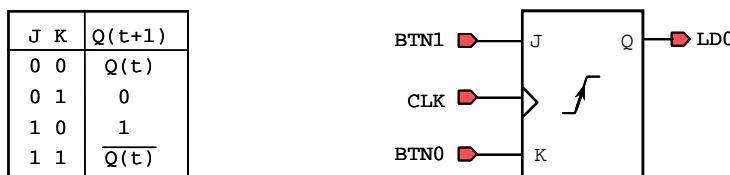


Figura 5.1: Tabla de verdad del biestable J-K (izquierda) y circuito que queremos implementar (derecha)

El código 5-1 muestra el proceso que implementa un biestable J-K activo por flanco de subida.

```
P_JK1: Process (Clk)
begin
  if Clk'event and Clk='1' then
    if J='1' and K='1' then
      Q <= not Q;
    elsif J='1' and K='0' then
      Q <= '1';
    elsif J='0' and K='1' then
      Q <= '0';
    else
      Q <= Q;
    end if;
  end if;
end process;
```

Código 5-1: Proceso que implementa un biestable J-K

```
P_JK2: Process (Clk)
begin
  if Clk'event and Clk='1' then
    if J='1' and K='1' then
      Q <= not Q;
    elsif J='1' and K='0' then
      Q <= '1';
    elsif J='0' and K='1' then
      Q <= '0';
    end if;
  end if;
end process;
```

Código 5-2: Otra alternativa para el biestable J-K

Ahora crea un proyecto llamado `biest_jk` y una nueva fuente VHDL del mismo nombre. Pon los puertos señalados en la figura 5.1, asignándole los pines que se muestran en la figura. Usa cualquiera de los procesos de los códigos 5-1 ó 5-2. Impleméntalo en la FPGA y comprueba que funciona correctamente.

Al implementarlo, no te olvides de indicar la frecuencia de reloj: En la ventana de *Processes*, dentro de *User Constraints*, pincha en *Create Timing Constraints*. Y allí saldrá la ventana de *Xilinx Constraints Editor*, pincha en la pestaña de *Global*.

Si estás usando la tarjeta *Pegasus* al reloj pon 20 en el periodo, que son 20 ns (50 MHz). El texto "ns" se añade automáticamente.

Si usas la tarjeta *Basys* el reloj se puede configurar a 100, 50 y 25 MHz según cómo coloques el *jumper J4*, el de configuración del reloj (figura 2.2).

Guarda y cierra el *Xilinx Constraints Editor*. La figura 5.2 muestra la ventana del Editor de Restricciones de Xilinx (*Xilinx Constraints Editor*), donde tenemos que indicar el periodo de reloj de la placa.

Una vez que lo tengas implementado, comprueba que funciona correctamente. Comprueba además qué sucede cuando mantienes presionados los dos pulsadores *PB0* y *PB1* simultáneamente. ¿Qué ocurre? ¿sabes por qué?

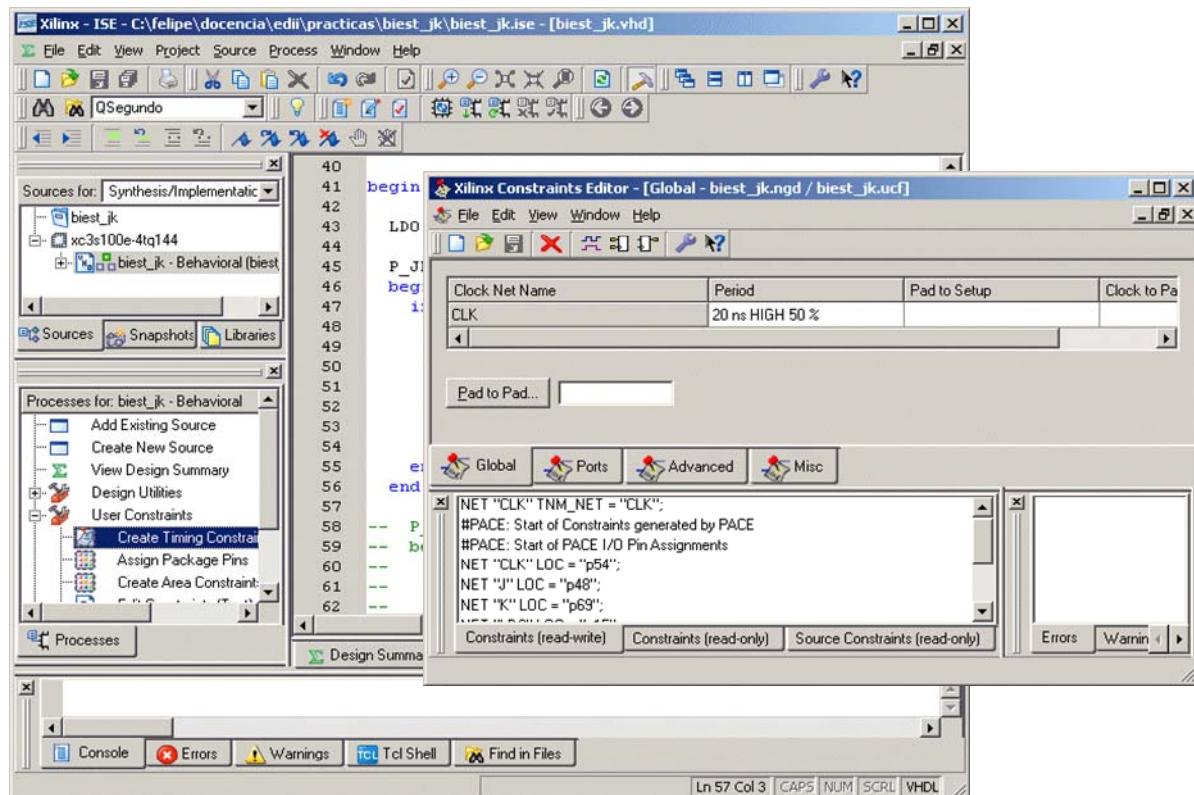


Figura 5.2: Editor de restricciones del ISE (Xilinx Constraints Editor)

## 5.2. Descripción de biestables en VHDL para síntesis

En este apartado analizaremos cómo se debe realizar la descripción de los biestables en VHDL para síntesis. La forma de describir biestables que se va a explicar no es la única admitida por el estándar de síntesis [14]. Sin embargo se te recomienda que sea esta forma la que utilices ya que es admitida por prácticamente todos los sintetizadores (por no decir todos). No siempre que se diga que es "incorrecto" significa que sea una forma incorrecta de describirlo, pero al menos, no se recomienda en esta asignatura, por claridad y uniformidad de código. Las reglas son:

- El proceso debe tener una sentencia que indica si es un elemento de memoria activo por nivel (*latch*) o por flanco (*flip-flop*). En el código 5-1, resaltado en negrita, indica que es un *flip-flop* activo por flanco de subida. los cuatro posibles casos son:
  - Si es un *flip-flop* activo por flanco de subida la sentencia de reloj debe ser:  
`if clk'event and clk='1' then`
  - Si es un *flip-flop* activo por flanco de bajada la sentencia de reloj debe ser:  
`if clk'event and clk='0' then`
  - Si es un *latch* activo por nivel alto la sentencia de reloj debe ser:  
`if enable='1' then`
  - Si es un *latch* activo por nivel bajo la sentencia de reloj debe ser:  
`if enable='0' then`
- La sentencia de reloj no debe de tener ninguna condición más a parte de la del reloj. Otras condiciones se deben de poner en otras sentencias *if* de menor jerarquía. El código 5-3 muestra la sentencia de reloj correcta, mientras que el código 5-4 es incorrecto porque la condición de *J='1'* está en la misma condición del reloj.

```
if clk'event and clk='1' then
  if J='1' then
    ....
```

```
if clk'event and clk='1' and J='1' then
  ....
```

Código 5-4: sentencia de reloj INCORRECTA

Código 5-3: Sentencia de reloj CORRECTA

- El biestable puede tener inicializaciones asíncronas (*reset*, *clear*, *set*, *preset*, ... o diversos nombres). Esta inicialización se debe poner antes de la sentencia de reloj. La sentencia de reloj se pone con un *elsif* que sigue a la sentencia de inicialización (código 5-5).

```
Biest_proc: Process (Reset, Clk)
begin
  if reset = '0' then
    Q <= '0';
  elsif clk'event and clk='1' then
    -- ahora van sentencias sincronas
  ....
```

Código 5-5: Inicialización síncrona

- Fuera de la sentencia de reloj no debe de haber ninguna sentencia más. Solamente se admite antes de la sentencia de reloj una inicialización asíncrona (como en el código 5-5), pero después no debería de haber sentencias, ni tampoco sentencias fuera de la inicialización. En los códigos 5-6 y 5-7 se muestran dos ejemplos en los que hay una sentencia de asignación antes de la inicialización síncrona (código 5-6), y una asignación fuera de la sentencia de reloj (código 5-7). Las sentencias incorrectas están en negrita.

```
Biest_proc: Process (Reset, Clk)
begin
  H <= P; -- INCORRECTO
  if Reset = '0' then
    Q <= '0';
  elsif Clk'event and Clk='1' then
    -- aquí van sentencias sincronas
  ....
```

Código 5-6: Asignación antes de la inicialización. INCORRECTA

```
Biest_proc: Process (Reset, Clk)
begin
  if Reset = '0' then
    Q <= '0';
  elsif Clk'event and Clk='1' then
    Q <= Dato;
  end if;
  H <= P; -- INCORRECTO
end process;
```

Código 5-7: Asignación fuera del reloj. INCORRECTA

- La sentencia de reloj debe de terminar con un *end if* y no debe de haber un *else* o *elsif* a continuación (en el mismo nivel del *if*). Puede haber sentencias *if* anidadas dentro de la sentencia de reloj (como en el código 5-1 y código 5-3), pero no al mismo nivel.

```

Biest_proc: Process (Reset, Clk)
begin
    if Reset = '0' then
        Q <= '0';
    elsif Clk'event and Clk='1' then
        Q <= Dato;
    else
        Q <= P;
    end if;
end process;

```

Código 5-8: Asignación después de la sentencia de reloj. INCORRECTA

- En la lista de sensibilidad del proceso debe aparecer la señal de reloj y todas las señales asíncronas que se leen (las que están antes de la señal de reloj). No es incorrecto incluir las señales síncronas (las que están después de la sentencia de reloj), pero incluirlas hace que la simulación vaya más lenta. El código 5-9 muestra un ejemplo. En él, la señal Dato no se incluye por ser una señal síncrona: está dentro de la condición de reloj. Incluirla no es incorrecto, pero hace la simulación más lenta.

```

Biest_proc: Process (Reset, ValorInicial, Clk)
begin
    if Reset = '0' then
        Q <= ValorInicial; -- dato asíncrono
    elsif Clk'event and Clk='1' then
        -- Aquí van las señales síncronas y no se ponen en la lista de sensibilidad
        Q <= Dato;
    end if;
end process;

```

Código 5-9: Lista de sensibilidad con el reloj y las señales asíncronas

- Dentro de la condición del reloj pueden ir todo tipo de sentencias. Estas sentencias son secuenciales y **sí importa el orden** de aparición. Si se asigna valor a una señal más de una vez, la última asignación es la que le dará valor. Así, los códigos 5-10 y 5-11 son equivalentes. El código 5-10 asigna `Q<='0'` cuando `Inic='1'`. Mientras que el código 5-11 asigna `Q<='0'` siempre, pero es sobrescrito por `Q<=Dato` cuando `Inic='0'`, por tanto, asigna `Q<='0'` cuando `Inic='1'`.

```

Biest_proc: Process (Reset, Clk)
begin
    if Reset = '0' then
        Q <= '0';
    elsif Clk'event and Clk='1' then
        if Inic = '0' then
            Q <= Dato;
        else
            Q <= '0';
        end if;
    end if;
end process;

```

Código 5-10: Proceso equivalente al del código 5-11

```

Biest_proc: Process (Reset, Clk)
begin
    if Reset = '0' then
        Q <= '0';
    elsif Clk'event and Clk='1' then
        Q <= '0';
        if Inic = '0' then
            Q <= Dato;
        end if;
    end if;
end process;

```

Código 5-11: Proceso equivalente al del código 5-10

- Cuando en un camino no se asigna ningún valor al biestable, éste guarda el valor que tenía antes. Por ejemplo, en el código 5-2, la señal `Q` no recibe valor cuando `J='0'` y `K='0'`, por tanto, la señal `Q` guarda su valor anterior. En consecuencia, el proceso del código 5-2 es equivalente al proceso del código 5-1. Puesto que en el código 5-1 se asigna `Q <= Q` cuando `J='0'` y `K='0'`.
- Esto último es muy importante para evitar crear un elemento de memoria cuando no queremos crearlo. Si queremos crear un proceso combinacional debemos incluir todas las posibilidades, o terminar el proceso con un `else` o un "when others". Otra alternativa es incluir una asignación por defecto al principio.

Por ejemplo, el código 5-12 no genera *latch*, porque con el `else` se cubren todas las posibilidades. Sin embargo, el proceso 5-13 sí genera *latch*. Aunque la señal `Select` nunca pudiera valer "11" se debe de poner según el código 5-12.

```
CombProc: Process(Select,Dato1,Dato2)
begin
  if Select = "00" then
    Q <= '0';
  elsif Select = "01" then
    Q <= Dato1;
  else
    Q <= Dato2;
  end if;
end process;
```

Código 5-12: Proceso combinacional (no genera latch). Correcto

```
LatchProc: Process(Select,Dato1,Dato2)
begin
  if Select = "00" then
    Q <= '0';
  elsif Select = "01" then
    Q <= Dato1;
  elsif Select = "10" then
    Q <= Dato2;
  end if;
end process;
```

Código 5-13: Proceso que genera latch.  
Posiblemente no lo quieras generar

Incluso, aunque se cubran todas las posibilidades se debe terminar con un `else` porque los tipos `std_logic` tienen más valores posibles aparte de '0' y '1'. El código 5-14 es el recomendado frente al código 5-15.

```
CombProc: Process(Select,Dato1,Dato2)
begin
  if Select = "00" then
    Q <= '0';
  elsif Select = "01" then
    Q <= Dato1;
  elsif Select = "10" then
    Q <= Dato2;
  else
    Q <= '1';
  end if;
end process;
```

Código 5-14: Proceso combinacional (no genera latch)

```
LatchProc: Process(Select,Dato1,Dato2)
begin
  if Select = "00" then
    Q <= '0';
  elsif Select = "01" then
    Q <= Dato1;
  elsif Select = "10" then
    Q <= Dato2;
  elsif Select = "11" then
    Q <= '1';
  end if;
end process;
```

Código 5-15: Proceso que es posible que genere latch. No recomendado

Otra manera de asegurarse de que no se genera *latch* es incluir una asignación por defecto al principio. Esto vendría a ser equivalente a un `else`. El código 5-16 es equivalente al código 5-12, y el código 5-17 es equivalente al 5-14.

```
CombProc: Process(Select,Dato1,Dato2)
begin
  Q <= Dato2;
  if Select = "00" then
    Q <= '0';
  elsif Select = "01" then
    Q <= Dato1;
  end if;
end process;
```

Código 5-16: Proceso combinacional equivalente al código 5-12 (no genera latch)

```
CombProc: Process(Select,Dato1,Dato2)
begin
  Q <= '1';
  if Select = "00" then
    Q <= '0';
  elsif Select = "01" then
    Q <= Dato1;
  elsif Select = "10" then
    Q <= Dato2;
  end if;
end process;
```

Código 5-17: Proceso combinacional equivalente al código 5-14 (no genera latch)

Por último, se recomienda indentar (incluir espacios o tabuladores) con cada sentencia anidada. Esto facilita mucho la comprensión del código. Se recomienda indentar con dos espacios. Para determinar la indentación con el ISE-WebPack, vete a *Edit→Preferences*, y allí busca la categoría *ISE Text Editor*. En ella pon *Tab Width* = 2. Y pon que cuando inserte un tabulador (*When You Press the Tab Key*) se pongan espacios: *Insert spaces*.

### 5.3. Encendido y apagado de LED con un pulsador

En la sección 5.1 vimos cómo se enciende y apaga un LED con dos pulsadores: uno para encender (*J*) y otro para apagar (*K*). Ahora queremos hacerlo con un sólo pulsador, de

modo que si está apagado, cuando pulsemos se encienda; y si está encendido, al pulsar se apague.

Sin mirar la solución que se da a continuación, intenta realizar el circuito tú sólo con la información de la sección 5.1.

### 5.3.1. Primera solución

La primera aproximación al problema sería realizar un biestable T, igual que como se hizo en la práctica 17 de ED1 [12]. Para recordar, en la figura 5.3 se muestra la tabla de verdad del biestable T y su esquema.

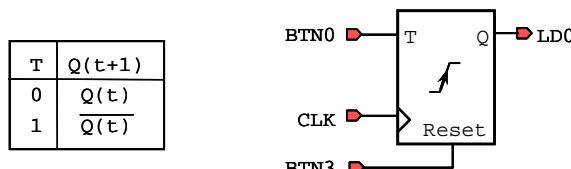


Figura 5.3: Tabla de verdad del biestable T y su esquema

Los códigos 5-18 y 5-19 muestran procesos equivalentes para implementar un biestable T activo por flanco de subida.

```
P_T: Process (Reset, Clk)
begin
    if Reset = '1' then
        Q <= '0';
    elsif Clk'event and Clk = '1' then
        if T = '1' then
            Q <= not Q;
        end if;
    end if;
end process;
```

Código 5-18: Proceso que implementa un biestable T

```
P_T: Process (Reset, Clk)
begin
    if Reset = '1' then
        Q <= '0';
    elsif Clk'event and Clk = '1' then
        if T = '1' then
            Q <= not Q;
        else
            Q <= Q;
        end if;
    end if;
end process;
```

Código 5-19: Alternativa que describe el biestable T

Ahora implementa este diseño en la FPGA, asociando el botón de encendido/apagado al pulsador `BTN0`, el `Reset` al pulsador `BTN3`, y la salida al LED `LD0`.

Comprueba varias veces si funciona bien y comprueba que sucede si mantienes presionado el botón de encendido/apagado. ¿Qué sucede? ¿funciona correctamente? ¿por qué?

### 5.3.2. Detector de flanco

¿Qué sucede? ¿Por qué a veces funciona el botón de encendido/apagado y otras veces se queda en el mismo estado en el que estaba? ¿Qué pasa si dejo el botón pulsado de manera continua?

En la figura 5.4 se muestra el cronograma de lo que sucede al pulsar el botón. Como la frecuencia del reloj es muy alta respecto a la velocidad humana, por muy rápido que pulsemos y soltemos el pulsador, el tiempo de pulsación durará muchos ciclos de reloj. Y por tanto, el estado del biestable ( $Q$ ) cambiará en todos los ciclos de reloj mientras esté el pulsador presionado. Al soltar el pulsador, el estado final será aleatorio, según se haya mantenido pulsado durante un número par o impar de ciclos.

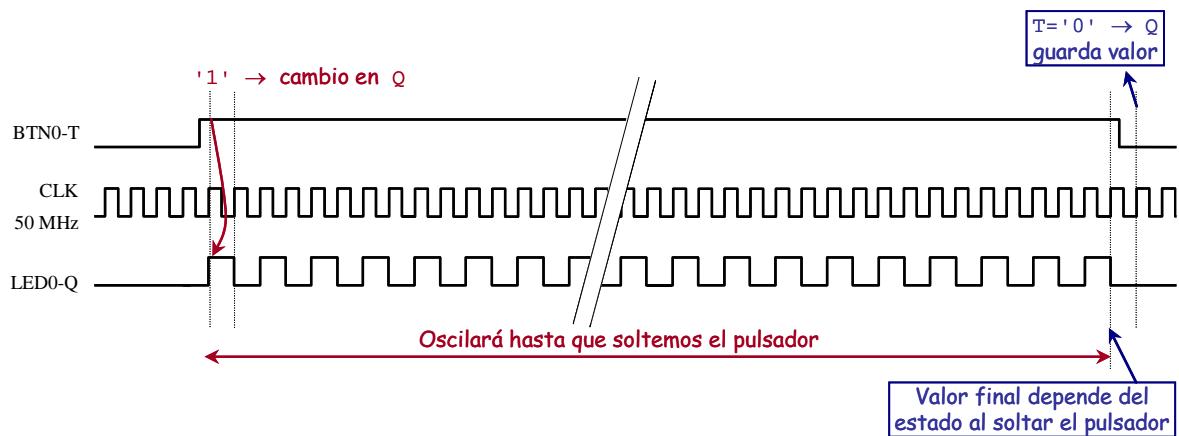


Figura 5.4: Cronograma resultante de pulsar el botón de encendido/apagado

Para evitar este efecto, lo que tenemos que hacer es un circuito detector de flancos, que transformará nuestro pulso humano en pulso del periodo del reloj. Podríamos pensar en utilizar la expresión **T'event and T='1'** para detectar el flanco de subida. Y seguramente funcione, sin embargo hemos visto que sólo debemos utilizar dicha expresión para la señal de reloj. Porque si no, estaremos usando la señal **T** como reloj del biestable, y en diseños síncronos no es recomendable. Así que tenemos que pensar en otra alternativa.

El detector de flanco se puede realizar con una máquina de estados que detecte la secuencia 0-1 para flancos de subida y la secuencia 1-0 para flancos de bajada.

La máquina de estados podemos hacerla mediante diagrama de estados o bien mediante registros de desplazamiento (usando biestables D). Ambos métodos se han visto en clase. Por ser una secuencia pequeña, la haremos mediante registros de desplazamiento. No usaremos directamente la entrada debido a que el pulsador es una entrada asíncrona, sino que la registraremos (la pasaremos por un biestable) para evitar metaestabilidad y para evitar que el pulso resultante dure muy poco. El circuito detector de flanco sin registrar la entrada se muestra en la figura 5.5 (realizado con registro de desplazamiento). Es una máquina de Mealy porque la salida depende de la entrada (**BTN0**).

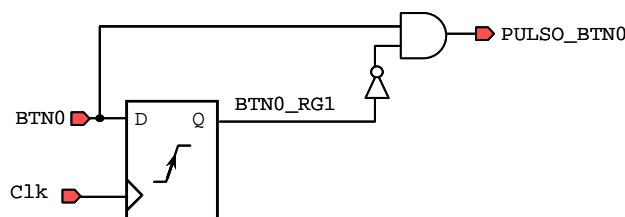


Figura 5.5: Circuito detector de flanco sin registrar la entrada (Mealy: no recomendado para este caso)

Sin embargo, el circuito de la figura 5.5 no es aconsejable porque la entrada **BTN0** es asíncrona, pues el pulsador puede ser presionado en cualquier momento. Esto puede ocasionar metaestabilidad o anchos de pulsos muy pequeño. El cronograma de este circuito para dos pulsaciones se muestra en la figura 5.6. En ella se puede ver que el segundo pulso está muy cerca del flanco del reloj, esto produce una salida muy estrecha, y además podría provocar metaestabilidad.

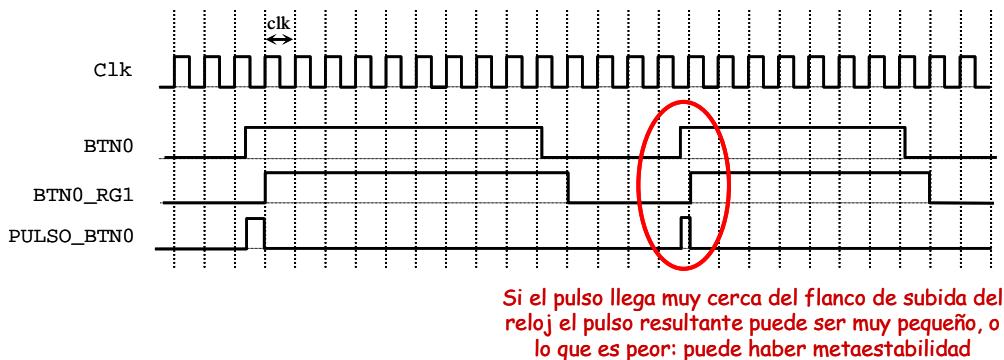


Figura 5.6: Cronograma del detector de flanco realizado por Mealy (no recomendado para este caso)

Para evitar estos inconvenientes se hace el registro de desplazamiento con la entrada registrada (máquina de Moore). La figura 5.7 muestra la versión Moore del circuito de detector de flanco realizada con registros de desplazamiento.

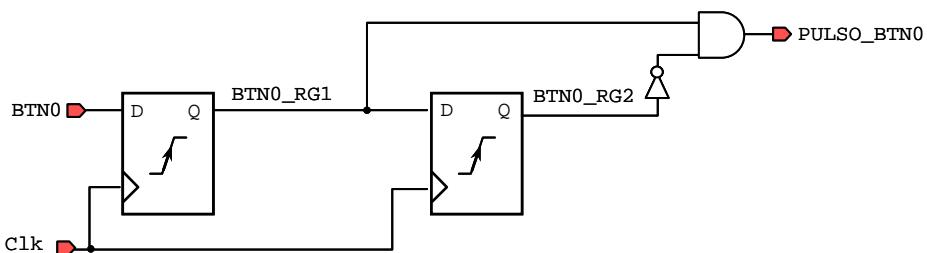


Figura 5.7: Circuito detector de flanco registrando la entrada (Moore: recomendado para este caso)

Ahora los pulsos resultantes siempre durarán un ciclo de reloj, como puedes ver en el cronograma de la figura 5.8.

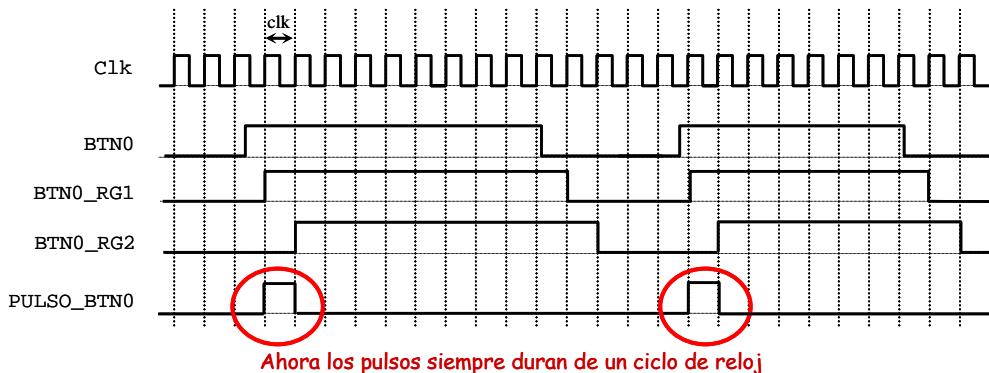


Figura 5.8: Cronograma del detector de flanco realizado por Moore (recomendado para este caso)

Así pues, el circuito que tenemos que hacer contendrá el biestable T de la figura 5.3, pero en vez de recibir el pulso directamente del pulsador, lo recibirá filtrado por el detector de flanco de la figura 5.7. El circuito final será el mostrado en la figura 5.9.

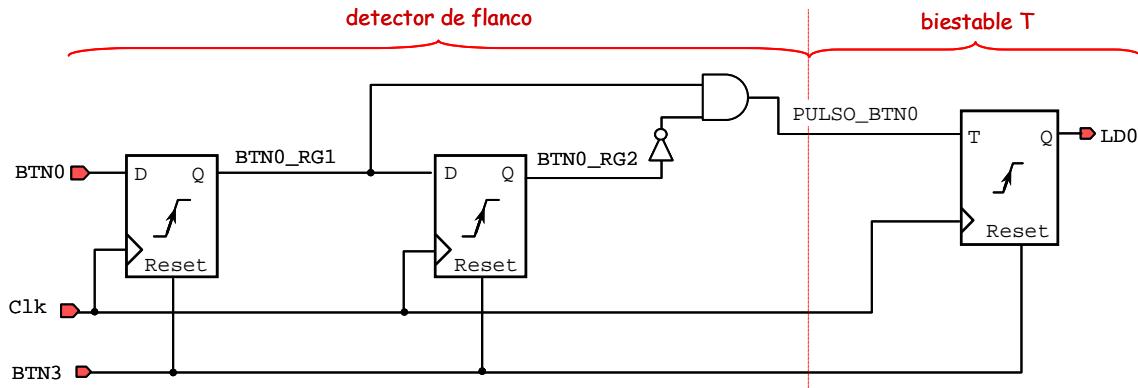


Figura 5.9: Circuito que enciende y apaga un LED con el mismo pulsador

Este circuito se puede describir con el código VHDL mostrado en la figura 5.10.

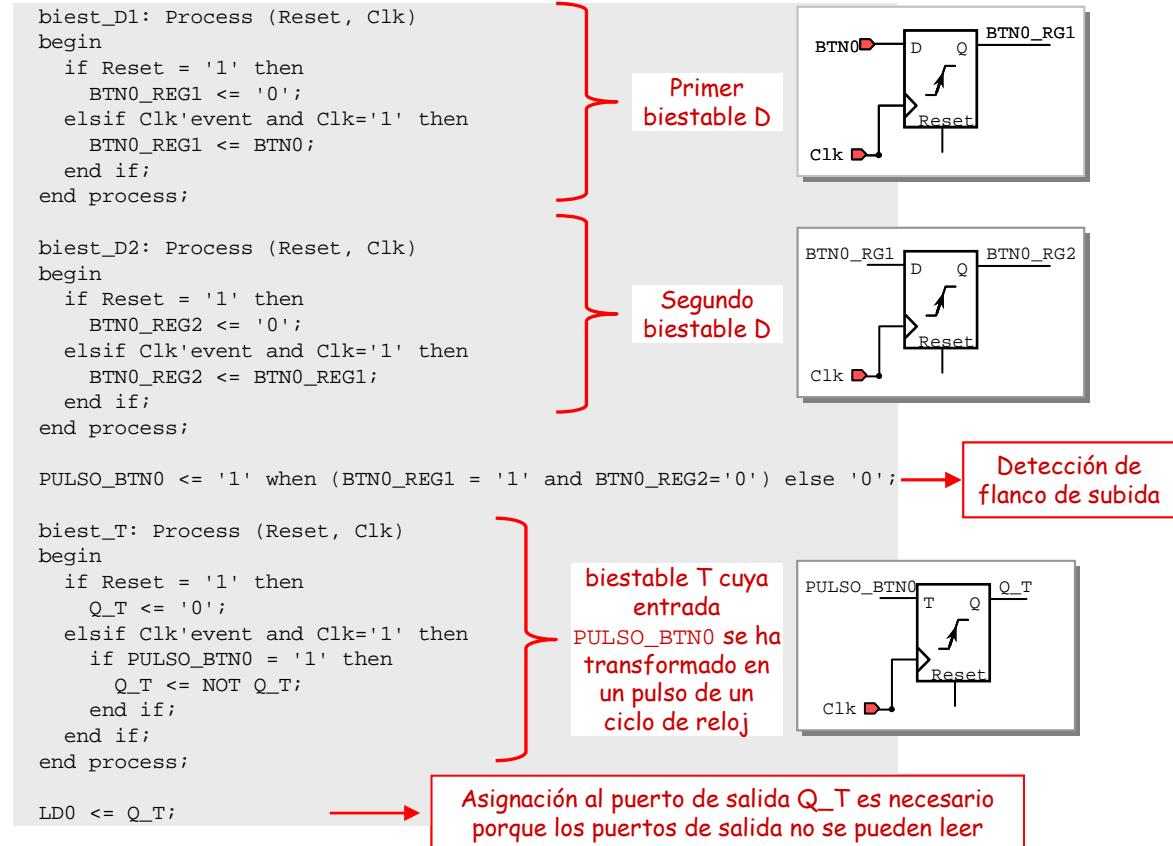


Figura 5.10: Código VHDL del circuito completo

Los dos primeros procesos de la figura 5.10 se pueden agrupar en uno, ya que **en los procesos el valor de las señales no se actualiza inmediatamente sino que se actualizan al terminar el proceso**. Así, en el código 5-20, la señal `BTN0_REG1` no toma el valor inmediatamente de `BTN0`, y por lo tanto, la señal `BTN0_REG2` no recibe el valor que se le acaba de asignar a `BTN0_REG1` (el que le da `BTN0`) sino el valor que tenía `BTN0_REG1`. En consecuencia, al salir el proceso, `BTN0_REG1` tendrá el valor de `BTN0` y `BTN0_REG2` tendrá el antiguo valor de `BTN0_REG1` y no el que se le acaba de asignar. Por tanto, `BTN0_REG2` está retrasada un ciclo de reloj respecto de `BTN0_REG1` (que es lo que queremos hacer).

Debido a esto, el proceso 5-21 es equivalente al proceso 5-20. Aunque las asignaciones estén cambiadas de orden.

```
biest_Ds: Process (Reset, Clk)
begin
    if Reset = '1' then
        BTN0_REG1 <= '0';
        BTN0_REG2 <= '0';
    elsif Clk'event and Clk='1' then
        BTN0_REG1 <= BTN0;
        BTN0_REG2 <= BTN0_REG1;
    end if;
end process;
```

Código 5-20: Proceso que implementa los 2 biestables D

```
biest_Ds: Process (Reset, Clk)
begin
    if Reset = '1' then
        BTN0_REG1 <= '0';
        BTN0_REG2 <= '0';
    elsif Clk'event and Clk='1' then
        BTN0_REG2 <= BTN0_REG1;
        BTN0_REG1 <= BTN0;
    end if;
end process;
```

Código 5-21: Proceso equivalente al 5-20

Ahora, crea un nuevo proyecto e implementa en la FPGA este circuito y comprueba que funciona bien. En caso de que estés trabajando con la placa *Basys*, el diseño funcionará mejor que el diseño de la figura 5.3, sin embargo, todavía no funcionará del todo bien porque los pulsadores de la *Basys* no tienen el circuito de eliminación de rebotes (figura 3.2) que sí está presente en la *Pegasus* (figura 3.1). Más adelante, en el capítulo 11 veremos cómo solucionar este problema.

En cualquier caso ¿qué pasa si en este circuito mantienes presionado el pulsador `BTN0`? ¿se comporta igual que antes?

## 5.4. Conclusiones

Los conceptos más importantes de esta práctica:

- No todas las maneras que tiene el VHDL para modelar elementos de memoria son válidas para síntesis.
- En esta práctica se han establecido unas reglas para describir elementos de memoria para síntesis.
- Cuando describimos **lógica combinacional**, hay que tener cuidado de no crear *latches no deseados*, para evitarlo, las señales se deben asignar en todas las condiciones posibles. Las condiciones deben terminar en `ELSE` y no `ELSIF`.
- En los procesos, las señales **no se actualizan inmediatamente** sino que se actualizan al terminar el proceso.
- Las entradas asíncronas se deben registrar para evitar **metaestabilidad**.
- Los **detectores de flancos** suelen ser necesarios para realizar interfaces humanos con diseños síncronos



## 6. Contadores

En esta práctica aprenderemos a describir contadores sintetizables en VHDL. Afortunadamente veremos que es mucho más sencillo que los contadores diseñados mediante esquemáticos que realizamos el curso pasado en ED1 [12]. En esta práctica veremos algunas aplicaciones de los contadores.

### 6.1. Segundero

El segundero que hicimos el año pasado y que tanto nos costó, este año lo podremos hacer en muy poco tiempo.

En esta práctica queremos que un LED se mantenga encendido durante un segundo y se apague durante el segundo siguiente, este comportamiento se repetirá periódicamente.

Recordando del año pasado, lo que tenemos que hacer es un divisor de frecuencia. Nuestro reloj tiene una frecuencia de 50 MHz y queremos una señal de 1 Hz (que llamaremos  $s_{1\text{seg}}$ ). El cronograma de estas señales se muestra en la figura 6.1. Como se puede apreciar, la señal  $s_{1\text{seg}}$  estará a '1' solamente durante un ciclo de reloj durante un segundo.

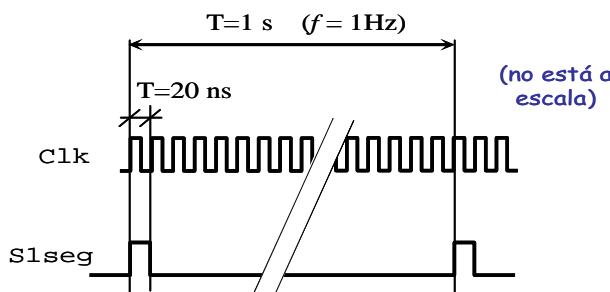


Figura 6.1: Cronograma de la señal que queremos obtener

La señal  $s_{1\text{seg}}$  será la entrada a un biestable T, y hará que cada segundo, el biestable cambie de valor. El esquema general del circuito se muestra en la figura 6.2.

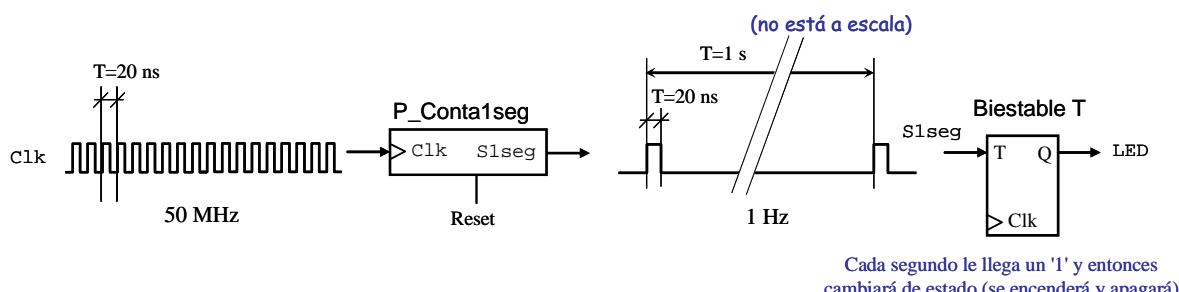


Figura 6.2: Esquema del circuito que queremos realizar

Para implementar un divisor de frecuencia necesitamos contar y para contar debemos ser capaces de sumar (al menos sumar 1 cada vez). Por otro lado, necesitamos saber qué rango tiene nuestra cuenta (hasta qué número llega) para que la señal que utilicemos para contar tenga un número adecuado de bits.

Para pasar de 20 ns a 1 segundo (de 50 MHz a 1 HZ), tenemos que contar hasta 50 millones ( $50 \cdot 10^6$ ). La menor potencia<sup>14</sup> de dos superior a  $50 \cdot 10^6$  es 26. Esto es,  $2^{26} > 50 \cdot 10^6 > 2^{25}$ . Por tanto, necesitamos 26 bits para poder representar el número  $50 \cdot 10^6$ . Así que nuestra señal de cuenta va a tener 26 bits.

Para llevar la cuenta se puede utilizar una señal de tipo entero o natural (integer o natural). Esta señal debe de tener un rango potencia de dos. Su declaración sería:

```
signal cuenta : natural range 0 to 2**26-1;
```

Fíjate que el rango de los enteros o naturales es ascendente (to en vez de downto como lo era en los std\_logic\_vector).

La operación potencia se representa con dos asteriscos en VHDL. Se debe restar uno al resultado de la potencia porque el rango empieza en cero. Por ejemplo para 8 bits, el rango sería:

```
signal cuenta256 : natural range 0 to 2**8-1; -- de 0 a 255
```

Si a un entero no se le especifica el rango, se genera una señal de 32 bits, lo que en la mayoría de los casos es un desperdicio de recursos.

La arquitectura completa del segundero se muestra en el código 6-1

```
architecture Behavioral of contalseg is
    signal cuenta : natural range 0 to 2**26-1;
    constant cfincuenta : natural := 50000000;
    signal slseg : std_logic;
    signal ledaux : std_logic;
begin
    -- Proceso que genera la señal periódica de 1 segundo
    P_contalseg: Process (reset, clk)
    begin
        if reset = '1' then
            cuenta <= 0;
            slseg <= '0';
        elsif clk'event and clk = '1' then
            if cuenta = cfincuenta-1 then      -- aquí se pone la constante en vez de 49999999
                cuenta <= 0;
                slseg <= '1';
            else
                cuenta <= cuenta + 1;
                slseg <= '0';
            end if;
        end if;
    end process;

    -- biestable T que cambia con la señal slseg, cada segundo
    P_LED: Process (reset, clk)
    begin
        if reset = '1' then
            ledaux <= '0';
        elsif clk'event and clk='1' then
            if slseg = '1' then
                ledaux <= not ledaux;
            end if;
        end if;
    end process;

    LD0 <= ledaux;
end Behavioral;
```

Código 6-1: Arquitectura del contador

<sup>14</sup> Para calcular el número de bits que necesitas, obtén el entero superior al resultado del logaritmo en base 2 del número. Por ejemplo,  $\log_2(50 \cdot 10^6) = 25,75 \rightarrow 26$  bits. Para calcular el logaritmo en base dos y no tienes en la calculadora, lo puedes calcular con el logaritmo en base 10:  $\log_2(X) = \log_{10}(X) / \log_{10}(2)$ . Otra forma (menos elegante) de hacerlo es ir calculando las potencias sucesivas de dos hasta encontrar la primera que sea mayor que el número.

Además del rango de la señal cuenta, fíjate en la constante `cfincuenta`. En vez de poner el número directamente en la expresión condicional, se pone una constante que hace más entendible el código.

Los dos procesos del código 6-1 se pueden unir en uno solo (código 6-2).

```
P_contalseg: Process (reset, clk)
begin
    if reset = '1' then
        cuenta <= 0;
        ledaux <= '0';
    elsif clk'event and clk = '1' then
        if cuenta = cfincuenta-1 then
            cuenta <= 0;
            ledaux <= not ledaux;
        else
            cuenta <= cuenta + 1;
        end if;
    end if;
end process;
```

Código 6-2: Proceso equivalente a los dos procesos del código 6-1

Asegúrate que entiendes porqué son equivalentes. Implementa cualquiera de las dos alternativas en la FPGA. Tendrás dos puertos de entrada (`reset` y `clk`) y uno de salida (`LED0`). Comprueba que el LED luce durante un segundo y se apaga en el siguiente. Y que sigue así sucesivamente.

## 6.2. Contador de 10 segundos

Ahora vamos a ampliar la práctica anterior y realizar un contador de 10 segundos que mostraremos en un *display* de siete segmentos. La cuenta irá de 0 a 9 (un dígito BCD).

Lo que haremos es utilizar la señal `s1seg` que creamos en el ejercicio anterior (figura 6.1 y código 6-1) para contar cada segundo. Así que utilizaremos dos contadores, uno que nos pasa de 50 MHz a 1 Hz y que crea la señal `s1seg`; y el otro contador contará diez cuentas de un segundo (`s1seg`). El esquema del circuito se muestra en la figura 6.3. A este esquema le falta el control de los ánodos para que luzca el *display* (repasa la práctica 4). Como puedes observar, la cuenta de 10 segundos se mostrará por un *display* de siete segmentos y por cuatro LED.

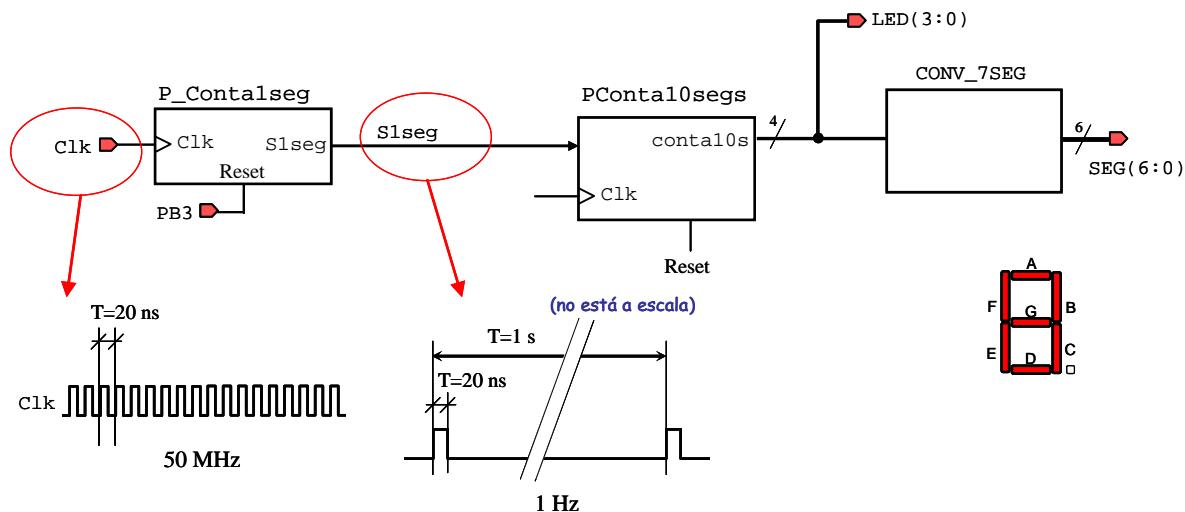


Figura 6.3: Esquema del contador de 10 segundos

En el ejercicio anterior vimos que podíamos utilizar números enteros o naturales para realizar contadores. Sin embargo, en VHDL para síntesis, el uso de números enteros y naturales no se recomienda cuando queremos mostrarlos por un *display* (o por cualquier otro medio). Esto se debe a que en los circuitos los números se representan en binario, y por tanto mejor es trabajar con vectores de bits. El número de bits del vector se corresponderá con los bits necesarios para representar el mayor y menor número que queramos representar.

Ya hemos visto vectores de bits: `std_logic_vector`. Sin embargo, sabemos del año pasado que la representación de un número cambia si utilizamos números sin signo (binario puro) o con signo (habitualmente complemento a 2). Por ejemplo, el número si la señal `S` = "1001", será el número 9 si es un número sin signo, o será el número -7 si es con signo en complemento a dos.

Así que para poder comparar, sumar y restar números binarios necesitamos saber si son números sin signo o si están en complemento a dos. Para ello se utilizan los tipos: `unsigned` y `signed`. Su declaración y manejo es similar al de los `std_logic_vector`. Por ejemplo, la declaración de la señal `conta10s`, que queremos que sea un número que cuente de 0 a 9, será:

```
signal conta10seg : unsigned (3 downto 0);
```

Código 6-3: Declaración de señal `unsigned`

Con esta declaración, especificamos un número binario de cuatro bits sin signo. Ya que el rango de un número de cuatro bits sin signo va de 0 a 15. Con tres bits no nos valdría porque sólo podríamos contar hasta siete, y por tanto, no sería un número BCD.

Para utilizar los tipos `unsigned` y `signed` es recomendable cambiar de bibliotecas, en vez de utilizar las predeterminadas por el *ISE-Webpack* (ver código 6-4), se recomiendan las del código 6-5. Esto está en la cabecera de todos los diseños que crea el *ISE-Webpack*. Así que a partir de ahora, siempre que crees una nueva fuente, quita las que están en el código 6-4 (las señaladas en negrita) y cámbialas por la del código 6-5 (la biblioteca `NUMERIC_STD`).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

Código 6-4: Bibliotecas por defecto que pone el ISE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

Código 6-5: Bibliotecas recomendadas

El proceso que se encarga de contar los diez segundos (`PConta10segs`) va a contar con la señal `conta10seg`, que es de tipo `unsigned` (ver código 6-3). El proceso se muestra en el código 6-6. Fíjate en la última sentencia: debido a que la señal `conta10seg` es de tipo `unsigned` y el puerto de salida `LED` es de tipo `std_logic_vector`, hay que convertir `conta10seg` a `std_logic_vector`. Para hacer esto, es necesario que ambas señales tengan el mismo número de bits (en este caso 4: 3 downto 0).

```

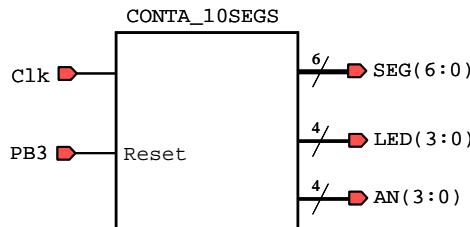
P_CONTA10SEG: Process (reset, clk)      -- contador de 10 segundos
begin
    if reset = '1' then
        conta10seg <= (others => '0');
        ledaux <= '0';
    elsif clk'event and clk='1' then
        if s1seg = '1' then -- contamos al
            if conta10seg = 9 then
                conta10seg <= (others => '0');
            else
                conta10seg <= conta10seg + 1;
            end if;
        end if;
    end process;

    -- hay que hacer un "cast" para convertir de unsigned a std_logic_vector
    LED <= std_logic_vector (conta10seg);

```

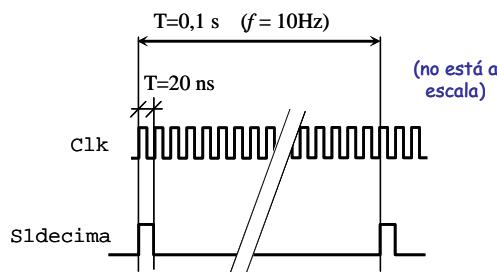
*Código 6-6: Proceso que cuenta 10 segundos*

Ahora implementa este circuito en la FPGA, las entradas y salidas del circuito se muestran en la figura 6.4

*Figura 6.4: Entradas y salidas del circuito*

### 6.3. Cronómetro

Ahora vamos a realizar un cronómetro digital que muestre décimas de segundos, unidades segundo, decenas de segundo y minutos. Para ello, haremos primero un divisor de frecuencia que nos obtenga una señal de décimas de segundo: `s1decima`, este divisor de frecuencia será similar al que generaba una de un segundo, pero la cuenta será 10 veces menor. La señal resultante, en vez de ser como la de la figura 6.1, será como la mostrada en la figura 6.5. Fíjate que la señal `s1decima` está a uno en un sólo ciclo de reloj durante todo su periodo de una décima de segundo.

*Figura 6.5: Cronograma de la señal de una décima de segundo que queremos obtener*

El proceso que implementa la señal `s1decima` es similar al proceso `P_Cuenta1seg` del código 6-1 (mira también la figura 6.2), pero la cuenta, en vez de ser de 50 millones será de 5000000 (diez veces menor).

A partir de la señal de una décima, realizaremos la cuenta de 10 décimas, 10 segundos, 60 segundos y 10 minutos. Utilizaremos cuatro procesos similares al `P_Cuenta10segs` (figura

6.3). De cada proceso obtendremos la cuenta en BCD (4 bits) y una señal que indicará el fin de cuenta para hacer contar al siguiente contador.

El esquema inicial del cronómetro se muestra en la figura 6.6. Le faltaría incluir la parte de visualización por los *displays* que la discutiremos más adelante.

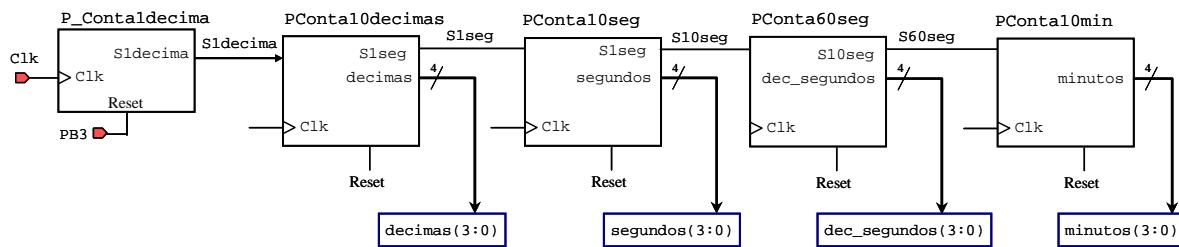


Figura 6.6: Esquema del cronómetro

Cada bloque de la figura 6.6 se puede implementar en un proceso. Ya dijimos que el primer proceso: `P_Contaldecima` es igual que el proceso `P_contalseg` del código 6-1, únicamente hay que cambiar el rango de la señal `cuenta` y el valor de la constante de fin de cuenta (ahora valdrá 5000000). ¿Cuál sería el rango de la señal `cuenta`?

El resto de procesos son similares al proceso `P_Conta10seg` del código 6-3, pero hay que cambiar los nombres de las señales y crear las señales de fin de cuenta: `S1seg`, `S10seg`, `S60seg`. Por ejemplo, el proceso `PConta10decimas` (ver figura 6.6), sería como el mostrado en el código 6-7 (la figura 6.7 muestra el esquema del contador de décimas). La cuenta de diez décimas de segundos produce la señal de aviso de que ha transcurrido un segundo: `s1seg`.

```

P_CONTA10DECIMAS: Process (reset, clk)
begin
    if reset = '1' then
        decimas <= (others => '0');
        s1seg <= '0';
    elsif clk'event and clk='1' then
        s1seg <= '0';
        if s1decima = '1' then
            if decimas = 9 then
                decimas <= (others => '0');
                s1seg <= '1';
            else
                decimas <= decimas + 1;
            end if;
        end if;
    end if;
end process;
    
```

Código 6-7: Proceso contador de décimas

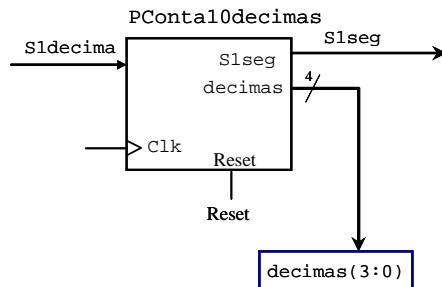


Figura 6.7: Esquema del proceso contador de décimas

Así que si has entendido todo lo que se ha explicado hasta aquí, no deberías de tener muchas dificultades para realizar el diseño mostrado en la figura 6.6. En caso de que no lo entiendas, repásate todas las prácticas, pregunta al profesor, o pide ayuda. Es fundamental que entiendas lo que se ha explicado hasta aquí para poder seguir con las prácticas siguientes que van aumentando en complejidad.

Ahora lo que nos queda es mostrar las cuatro cifras BCD por los cuatro *displays*. Sin embargo, hemos visto que en la placa sólo tenemos un grupo de siete pines para los cuatro *displays*. También tenemos cuatro ánodos para controlar qué *display* se tiene que encender. Entonces, ¿cómo podremos mostrar las cuatro cifras BCD? Veremos una primera aproximación manual (solución manual) y luego la solución automática.

### 6.3.1. Mostrar los dígitos. Solución manual

Para ver los cuatro dígitos podemos hacer un decodificador como lo hicimos en el apartado 4.2 (práctica 4). Según la codificación de los interruptores SW6 y SW7 se encenderá un *display* distinto. Esto es, se activará un ánodo distinto. Será parecido al código 4-1 pero sin la habilitación. La tabla 6-1 muestra los *displays* que se encienden y el número que se muestra según los interruptores que estén activados.

SW7	SW6	Ánodo	DISPLAY
0	0	AN3	décimas
0	1	AN2	segundos
1	0	AN1	decenas de segundos
1	1	AN0	minutos

Tabla 6-1: Displays que se muestran según la configuración de los interruptores

La figura 6.8 muestra el decodificador para activar los ánodos. Este decodificador es similar al del apartado 4.2 (bloque de abajo de la figura 4.6).

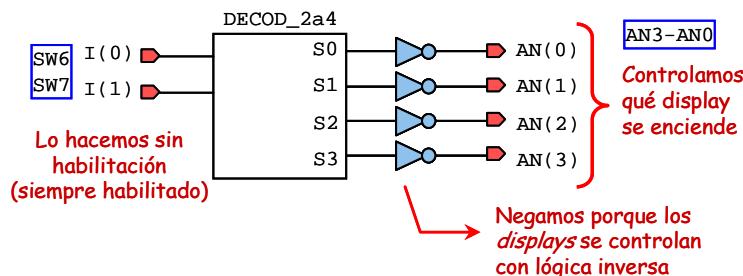


Figura 6.8: Codificador para controlar los ánodos de los displays

Además habrá que seleccionar la cifra que se quiere mostrar (décimas, unidades de segundo, decenas de segundo o minutos). La selección de la cifra se hará conforme a la tabla 6-1. Como ya sabemos, para seleccionar necesitamos un multiplexor. Después del multiplexor, pondremos un conversor a siete segmentos para mostrarlo por los *displays*. La figura 6.9 muestra el esquema de esta parte del diseño.

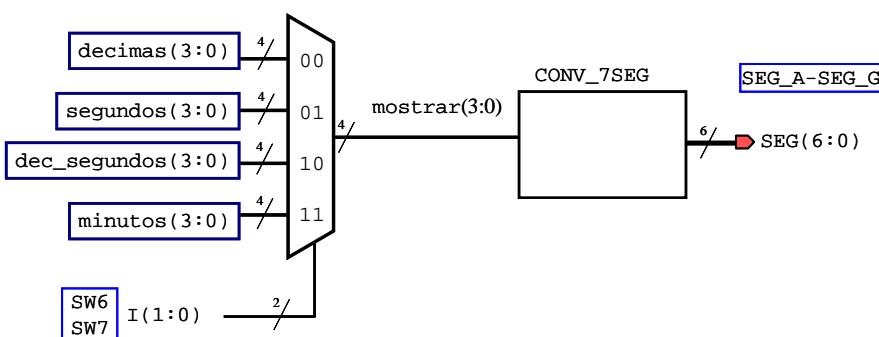


Figura 6.9: Multiplexor que selecciona la cifra BCD según la configuración de los interruptores.

El esquema completo del diseño se muestra en la figura 6.10.

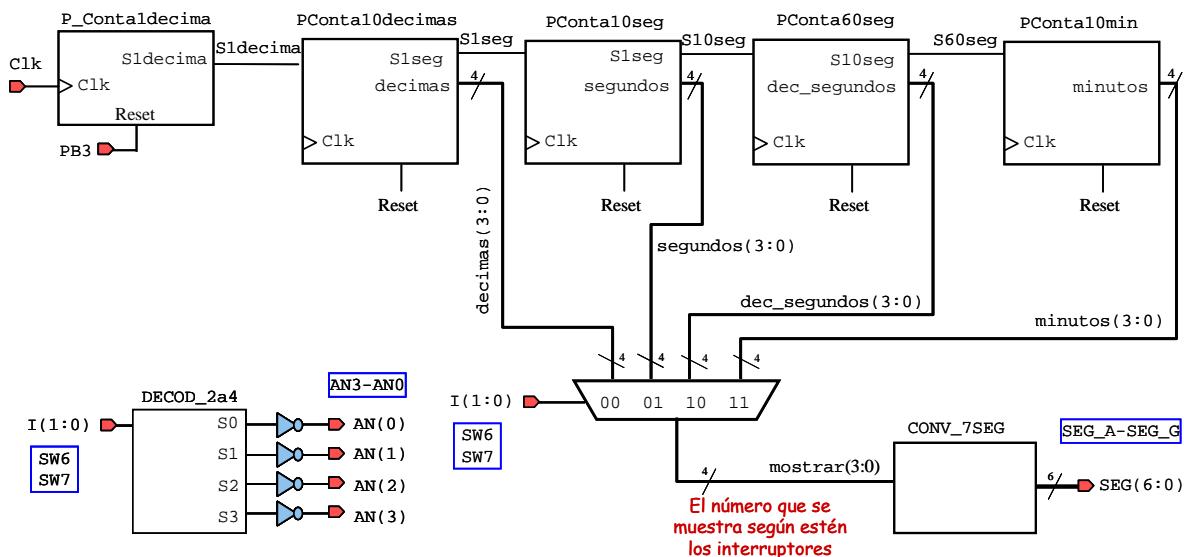


Figura 6.10: Esquema completo del cronómetro con visualización manual

Ahora crea un nuevo proyecto llamado `crono_manual`, implementa el diseño de la figura 6.10 en la FPGA y comprueba que funciona. Evidentemente, es un cronómetro de prestaciones muy malas, porque no puedes visualizar simultáneamente más de un cifra. Tienes que seleccionar la cifra manualmente con los interruptores `SW6` y `SW7`.

### 6.3.2. Mostrar los dígitos. Solución automática

Hemos visto las limitaciones de la solución anterior. Si quisieras ver dos cifras simultáneamente, podrías probar a mover el interruptor manualmente lo más rápidamente posible. Es obvio que esta solución no es muy cómoda, así que vamos a intentar buscar una alternativa.

La solución podría ser que en vez de que nosotros cambiamos la señal `I(1:0)` moviendo los interruptores `SW6` y `SW7`, hiciésemos un circuito que haga como si los moviésemos periódicamente de manera automática. La velocidad del cambio debe ser tan rápida que no sea perceptible por nosotros los humanos. Por ejemplo, si cambiamos el *display* cada milisegundo seguramente no nos demos cuenta de que está cambiando, salvo que como luce menos tiempo, lo veremos lucir con menor intensidad.

Así que vamos a probar de esta manera: crearemos otro contador similar al que genera una señal de periódica de un segundo (proceso `P_contalseg` del código 6-1) o similar al que hemos creado en el cronómetro para contar una décima de segundo (figura 6.6). Nuestro contador creará una señal periódica de un milisegundo (`s1mili`) como la mostrada en la figura 6.11.

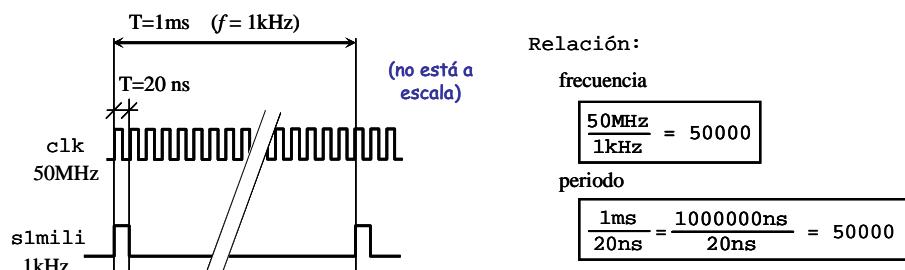


Figura 6.11 : Cronograma de la señal de una milésima de segundo que queremos obtener

A partir de la señal de un milisegundo (`s1mili`) realizaremos una cuenta de cuatro (de cero a tres) para seleccionar el *display* que va a lucir y el número que se va a mostrar. Esta señal (`CUENTA4MS`) tendrá una funcionalidad equivalente a la señal `i` que antes era gobernada por los interruptores (ver figuras 6.8, 6.9 y 6.10).

En la figura 6.12 se muestra el multiplexado en el tiempo. En cada milsegundo se muestra un *display* distinto (sólo hay un ánodo activo). La frecuencia de refresco del *display* es de 250 Hz ( $T=4\text{ms}$ ). Es muy importante seleccionar el número correspondiente al ánodo que se va a seleccionar.

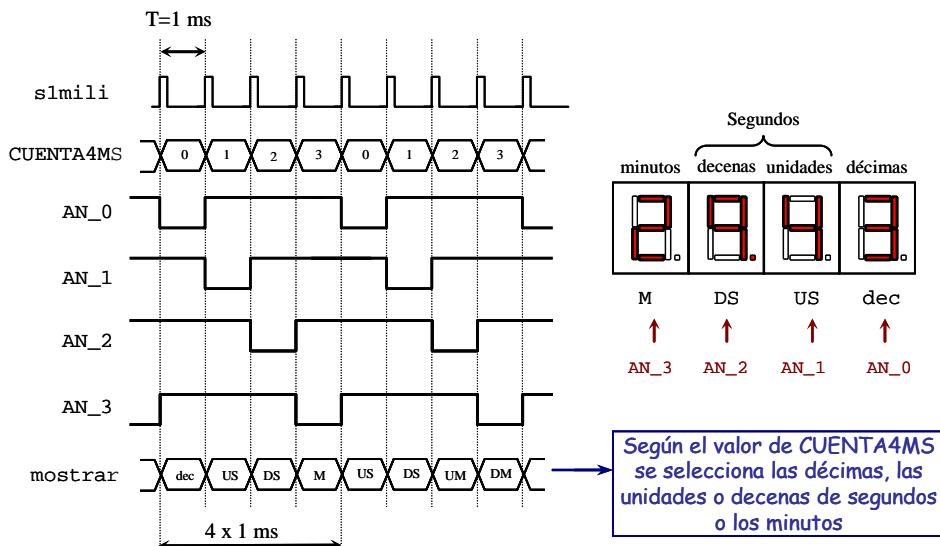


Figura 6.12: Multiplexado en el tiempo para mostrar los displays

El esquema del circuito final se muestra en la figura 6.13. Crea un nuevo proyecto llamado `crono_auto` e impleméntalo en la FPGA. Comprueba que te sale bien.

Como conclusión, hemos aprendido a hacer un reloj digital. Hemos visto cómo podemos multiplexar en el tiempo una varias señales. Con esto reducimos el número de pines de la FPGA para manejar los *displays*. Con esta solución tenemos los siete segmentos más el punto decimal (8 pines) más los cuatro ánodos, en total 12 pines. De la otra forma, si necesitásemos los 8 pines por cada *display* sería  $8 \times 4 = 32$  pines más los cuatro ánodos: 36 pines.

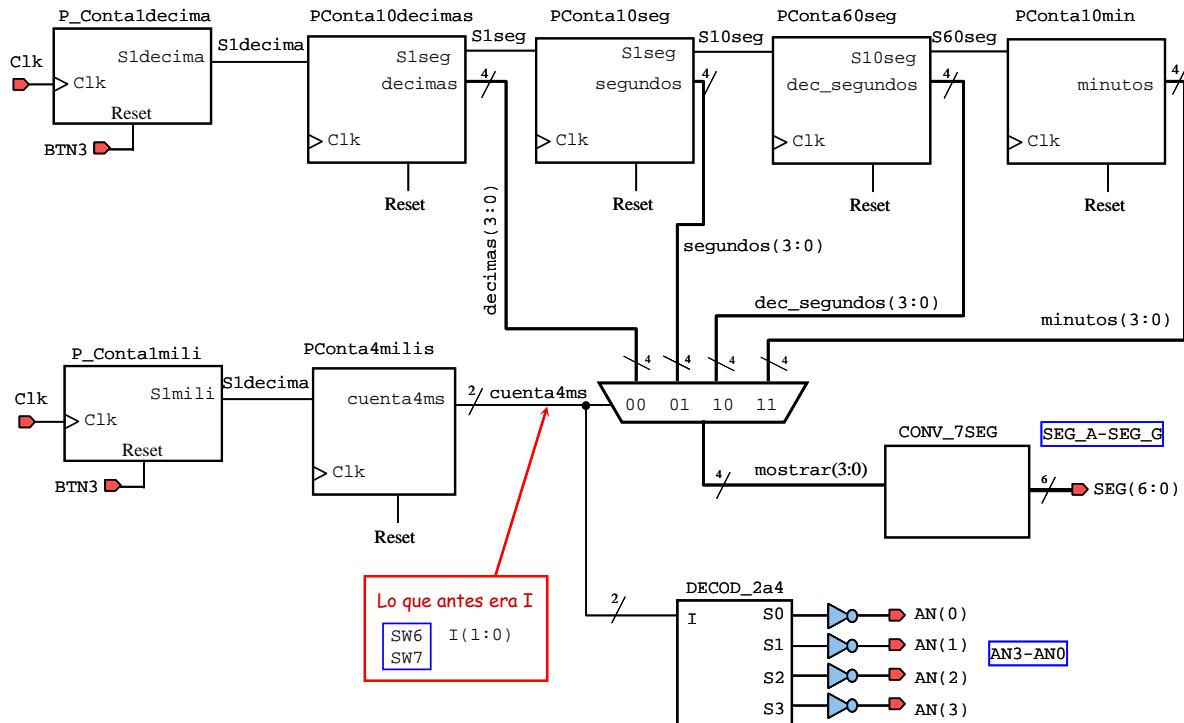


Figura 6.13: Esquema completo del cronómetro con visualización automática

### 6.3.3. Mejoras del circuito

Si te apetece puedes probar a introducir las siguientes mejoras:

- Muestra el punto decimal en el segundo y cuarto *display* para separar los segundos de las décimas de segundo y los minutos.
- Haz que el circuito se pueda parar con un interruptor. El cronómetro se debe mantener parado en el tiempo en que estaba.
- Haz que el circuito se pueda parar con un pulsador. Si pulsamos, el cronómetro se mantiene parado en el tiempo en que estaba, y si volvemos a pulsar, el cronómetro continúa.
- Haz un reloj de minutos y horas, haciendo que el punto decimal del medio parpadee cada segundo.

### 6.3.4. Optimización

Sabemos que hay muchas formas posibles de realizar un circuito. En concreto para este circuito nos podemos plantear poner el multiplexor antes o después de haber convertido el número a siete segmentos. ¿Qué manera crees que es más óptima en cuanto a ahorro de recursos de la fpga?

En la figura 6.14 se muestra cómo sería este circuito. Este circuito es una alternativa más costosa en términos de recursos respecto al circuito de la figura 6.9. Tanto porque usa más convertidores, como porque el multiplexor tiene un ancho de bus de los datos mayor (7 bits frente a 4). Al diseñar en VHDL debes tener muy presente qué circuito estás generando para evitar desperdiciar recursos.

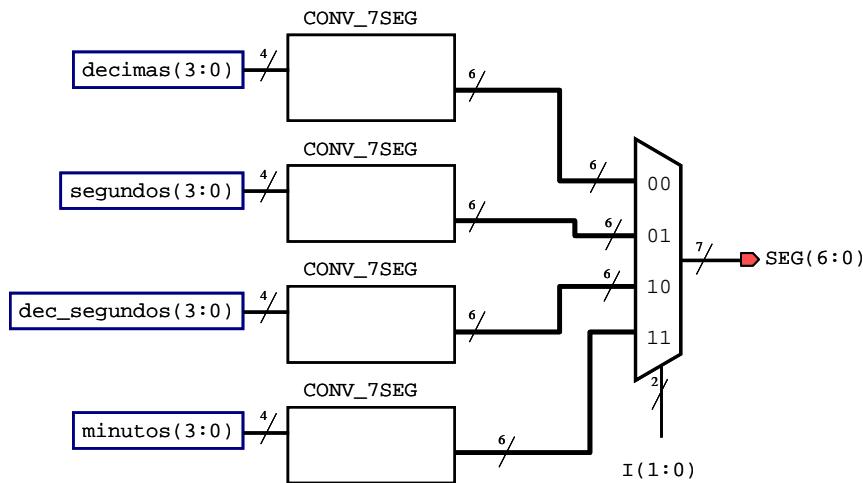


Figura 6.14: Otra alternativa al esquema de la figura 6.9

#### 6.4. Contador manual

Ahora vamos a realizar un contador ascendente/descendente que aumentará su cuenta cuando presionemos el pulsador `BTN0` (`UP`) y que disminuirá su cuenta cuando presionemos el pulsador `BTN1` (`DOWN`). Cuando no se presionen estos pulsadores el contador permanecerá quieto (no aumentará con el reloj de la placa como los de las prácticas anteriores).

La cuenta irá de 0 a 7 (3 bits), y no se desbordará, esto es, si pulsamos el `UP` estando en siete se quedará fijo (no pasa a cero), y si le damos al `DOWN` estando en cero, se quedará en cero (no pasa a siete).

El reset del circuito irá al pulsador `BTN4`.

El diagrama de estados del contador se muestra en la figura 6.15.

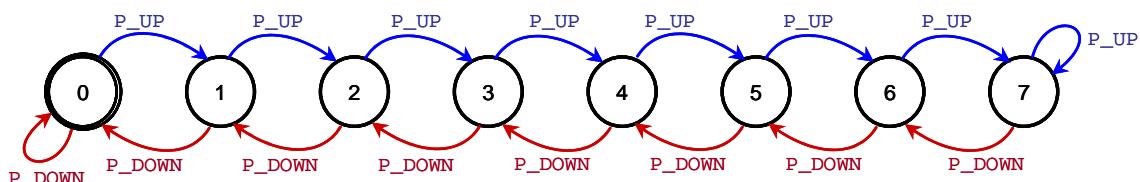


Figura 6.15: Diagrama de estados del contador ascendente/descendente que vamos a realizar

Para que cada vez que presionemos los pulsadores sólo aumente una cifra la cuenta, debemos realizar un detector de flanco para cada pulsador. Por eso, en la figura 6.15 las señales no son `UP` y `DOWN` sino `P_UP` y `P_DOWN`, para indicar que se ha convertido la entrada en un pulso y sólo está activa durante un ciclo de reloj. Si te has olvidado de cómo se hace el detector de flanco, consulta el apartado 5.3.2.

La cuenta la vamos a mostrar por el *display* de la derecha. El esquema del circuito se muestra en la figura 6.16. El bloque de la izquierda, podrá estar formado por un proceso y varias sentencias concurrentes, para simplificar, como ya sabemos cómo se hace, se ha puesto en un sólo bloque.

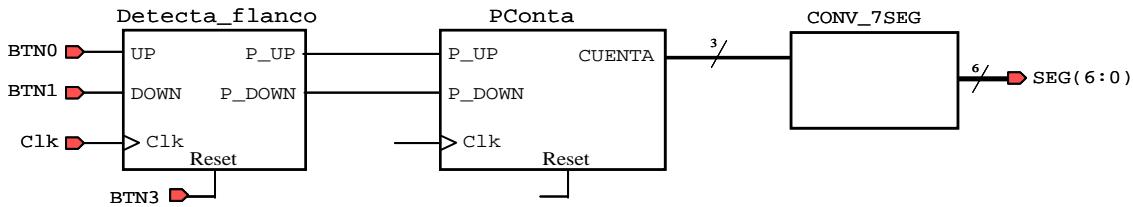


Figura 6.16: Esquema del contador que implementaremos

Intenta realizar el contador tú sólo (proceso `PConta` de la figura 6.16). Si no te sale, consulta el código 6-8

```

P_conta: Process (reset, clk)
begin
    if reset = '1' then
        conta <= (others => '0');
    elsif clk'event and clk='1' then
        if pulso_up = '1' then
            if conta /= 7 then
                conta <= conta + 1;
            end if;
        elsif pulso_down = '1' then
            if conta /= 0 then
                conta <= conta - 1;
            end if;
        end if;
    end process;
    
```

Código 6-8: Código del contador ascendente/descendente que no se desborda

Ahora implementa el circuito en la FPGA. y comprueba que funciona correctamente.

## 6.5. Conclusiones

Ahora resaltaremos los conceptos de esta práctica:

- Los divisores de frecuencia se implementan con contadores
- Las señales de tipo entero o natural (`integer` o `natural`) deben de tener un rango potencia de dos. Por ejemplo: `singal conta : integer range 0 to 2**8-1;`. El rango de esta señal es de 0 a 255, usando por tanto 8 bits (los dos asteriscos es la operación potencia en VHDL).
- Si a un entero o natural no se le especifica el rango, se genera una señal de 32 bits, lo que en la mayoría de los casos es un desperdicio de recursos.
- Los tipos `integer` y `natural` se pueden utilizar para realizar cuentas internas, pero para realizar cuentas que van a ser utilizadas por otros procesos o bloques, se recomienda utilizar los tipos vectoriales: `unsigned` y `signed`. Para representar números es importante utilizar estos tipos en vez del tipo `std_logic_vector`, para distinguir si la representación del número es en binario puro (sin signo) o en complemento a dos (con signo).
- Al usar los tipos `unsigned` y `signed` se recomienda utilizar la biblioteca `NUMERIC_STD` (el código 6-5 en vez del código 6-4).
- Multiplexar en el tiempo nos puede hacer ahorrar enormemente el número de puertos de salida (pines) necesarios.
- Distintas maneras de describir un circuito pueden hacer que consuma más o menos recursos de la FPGA. La optimización del circuito pretende reducir el uso de recursos. Existen diversos objetivos en la optimización, la optimización puede estar orientada al área, consumo, tiempos, ...

## 7. Registros de desplazamiento

En esta práctica aprenderemos a diseñar registros de desplazamiento en VHDL. Hay varios tipos de registros de desplazamiento, según sea la entrada en serie o paralelo, y la salida en serie o paralelo. También se pueden desplazar a la izquierda o a la derecha, o también hacer rotaciones (el bit que sale entra por el otro extremo). Vamos a irlos viendo con ejemplos.

### 7.1. Registro de desplazamiento con carga paralelo

En este primer ejemplo vamos a realizar un registro de desplazamiento con las siguientes características:

- El registro tendrá 8 bits
- Desplazamiento a la izquierda. La orden de desplazamiento la dará el pulsador **BTN2**. Habrá que realizar un detector de flanco para este pulsador.
- Al desplazar a la izquierda, lo que haya en el bit 7 del registro se perderá, y lo que haya en el bit cero se mantendrá (no se le asigna ningún valor nuevo).
- Con carga paralelo. La orden de cargar dato la dará el pulsador **BTN0**. Habrá que realizar un detector de flanco para este pulsador.
- Con la orden de carga, el registro se cargará con los datos proporcionados por los interruptores. Esto es, si **SW0** está a uno, se cargará un '1' en el bit 0 del registro.
- El reset será el pulsador **BTN3**.
- El valor de los bits del registro se mostrarán por los LED.

En la figura 7.1 se muestra el esquema del registro que queremos realizar.

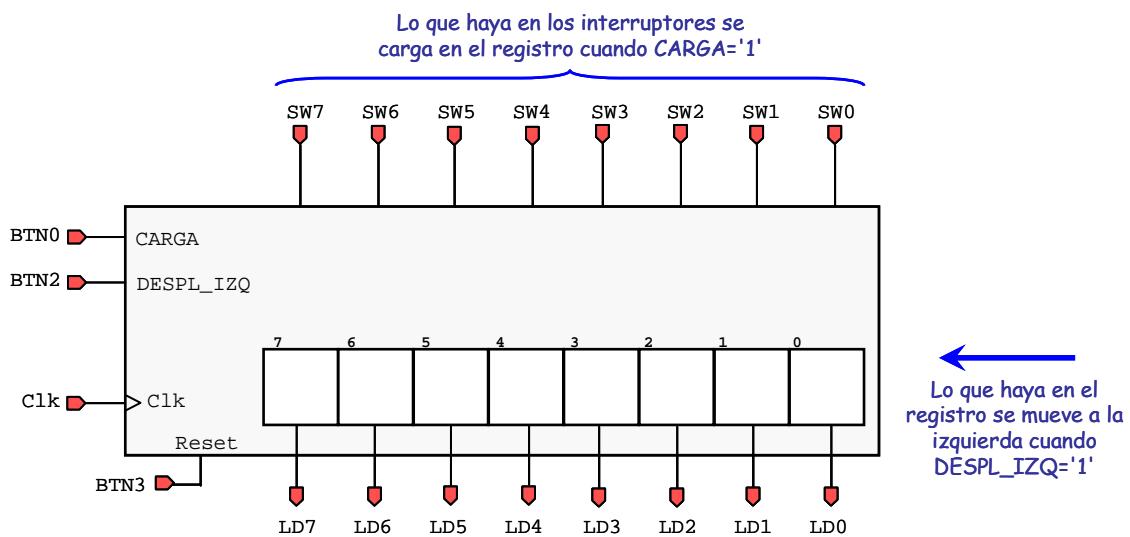


Figura 7.1: Esquema del registro de desplazamiento a la izquierda con carga paralelo

El proceso VHDL del registro de desplazamiento se muestra en el código 7-1.

```

P_registro: Process (reset, clk)
begin
    if reset = '1' then
        registro <= (others => '0');
    elsif clk'event and clk='1' then
        if pulso_carga = '1' then
            registro <= dato;
        elsif pulso_desplz_izq = '1' then
            registro (7 downto 1) <= registro (6 downto 0); -- desplazamiento a izquierda
        end if;
    end if;
end process;

```

Código 7-1: Código del proceso de registro de desplazamiento

Si has entendido las prácticas anteriores, no deberías de tener problemas para entender el código 7-1, salvo para la sentencia resaltada en negrita.

Esta sentencia sería equivalente a las señaladas en negrita en los códigos 7-2 y 7-3

```

P_registro: Process (reset, clk)
begin
    if reset = '1' then
        registro <= (others => '0');
    elsif clk'event and clk='1' then
        if pulso_carga = '1' then
            registro <= dato;
        elsif pulso_desplz_izq = '1' then
            registro (7) <= registro (6);
            registro (6) <= registro (5);
            registro (5) <= registro (4);
            registro (4) <= registro (3);
            registro (3) <= registro (2);
            registro (2) <= registro (1);
            registro (1) <= registro (0);
        end if;
    end if;
end process;

```

Código 7-2: Código equivalente al código 7-1

```

P_registro: Process (reset, clk)
begin
    if Reset = '1' then
        registro <= (others => '0');
    elsif clk'event and clk='1' then
        if pulso_carga = '1' then
            registro <= dato;
        elsif pulso_desplz_izq = '1' then
            registro (1) <= registro (0);
            registro (2) <= registro (1);
            registro (3) <= registro (2);
            registro (4) <= registro (3);
            registro (5) <= registro (4);
            registro (6) <= registro (5);
            registro (7) <= registro (6);
        end if;
    end if;
end process;

```

Código 7-3: Código equivalente al código 7-1

Es muy importante recordar que las señales dentro de un proceso no se actualizan hasta el final del proceso, así que, a pesar de que en el código 7-3 se podría pensar que el `registro(0)` se propagaría inmediatamente a todos los demás bits, no ocurre porque el `registro(1)` no toma el valor de `registro(0)` hasta que se sale del proceso.

Lo que ocurre al desplazar a la izquierda se muestra en la figura 7.2.

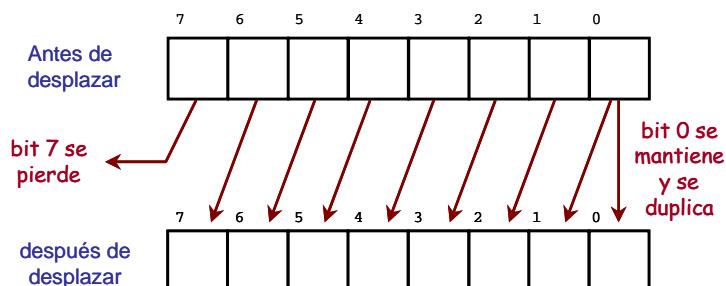


Figura 7.2: Desplazamiento a la izquierda

Ahora implementa el circuito y comprueba que funciona correctamente. Realiza la carga y luego el desplazamiento. Realiza varias cargas en las que unas veces el bit 0 valga '0' y otras '1'. Analiza lo que ocurre y asegúrate de que lo entiendes.

## 7.2. Rotación a la derecha y a la izquierda

Ahora en vez de perder el bit de un extremo y repetir el del otro (figura 7.2). Vamos a introducir el bit saliente en el otro extremo. Esto es una rotación. Además, realizaremos la rotación en ambos sentidos. BT2 rotará a la izquierda y BTN rotará a la derecha. La carga se realizará igual que en la práctica anterior.

El funcionamiento de la rotación a la izquierda se muestra en la figura 7.3

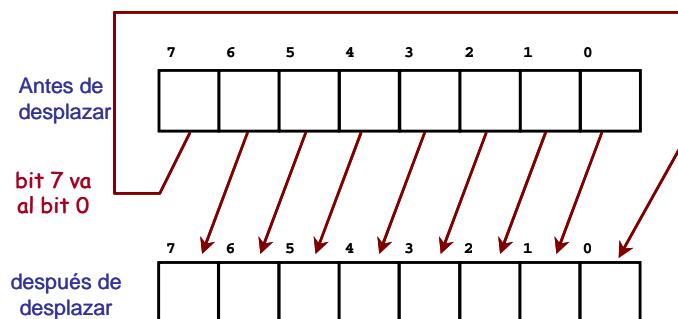


Figura 7.3: Rotación a la izquierda

Este proceso se muestra en el código 7-4.

```
P_registro: Process (reset, clk)
begin
    if reset = '1' then
        registro <= (others => '0');
    elsif clk'event and clk='1' then
        if pulso_carga = '1' then
            registro <= dato;
        elsif pulso_desplz_izq = '1' then
            registro <= registro (6 downto 0) & registro(7); -- rotacion izquierda
        elsif pulso_desplz_dcha = '1' then
            registro <= registro(0) & registro (7 downto 1); -- rotación derecha
        end if;
    end if;
end process;
```

Código 7-4: Código del proceso que rota a izquierda y derecha

En negrita están señaladas las sentencias de rotación. En ella aparece el operador concatenación: &. Con este operador se unen bits o vectores, en vectores de mayor tamaño.

Por ejemplo, el código 7-4 es equivalente al código 7-5:

```
P_registro: Process (Reset, Clk)
begin
    if Reset = '1' then
        registro <= (others => '0');
    elsif Clk'event and Clk='1' then
        if pulso_carga = '1' then
            registro <= dato;
        elsif pulso_desplz_izq = '1' then
            registro (7 downto 1) <= registro (6 downto 0) -- rotacion izquierda
            registro (0) <= registro(7);
        elsif pulso_desplz_dcha = '1' then
            registro (7) <= registro(0); -- rotación derecha
            registro (6 downto 0) <= registro (7 downto 1);
        end if;
    end if;
end process;
```

Código 7-5: Código equivalente al código 7-4

Ahora implementa este diseño en la FPGA y comprueba que rota a izquierda y derecha el dato que cargas en paralelo.

### 7.3. Rotación automática

Vamos a combinar el ejercicio de los contadores con los registros. Ahora, en vez de indicar nosotros cuando se realiza la rotación, vamos a hacer que se lleve a cabo periódicamente de forma automática.

Para ello vamos a realizar un divisor de frecuencia que dé una señal (`s1decima`) con una frecuencia de una décima de segundo. Una señal igual que la que generamos en el cronómetro (apartado 6.3, figura 6.5). Esta señal va a ordenar la rotación, y no como antes, que lo hacíamos con el pulsador. La rotación se hará a la izquierda.

En este ejercicio no realizaremos la carga, sino que al resetear cargaremos el registro con el valor "`00000001`", de modo que haya un sólo LED encendido. El esquema del circuito se muestra en la figura 7.4.

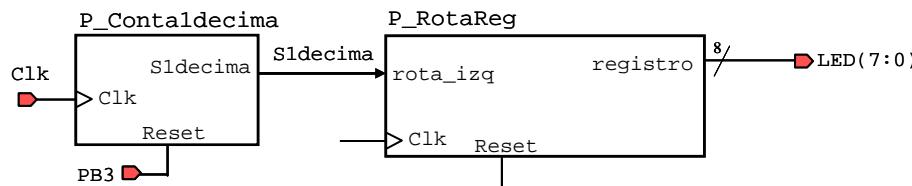


Figura 7.4: Esquema del circuito de rotación automática

Implementa este circuito y comprueba que funciona correctamente.

#### 7.3.1. Variantes del circuito

Puedes realizar variantes de este circuito:

- El sentido de la rotación (izquierda o derecha) se haga según como esté el primer interruptor, `SW0`
- El segundo interruptor hace que haya rotación o se quede el registro quieto.

### 7.4. Conclusiones

Lo principal de este capítulo es que hemos aprendido a implementar registros de desplazamiento en VHDL, repasando el concepto de que el valor de las señales se actualiza al salir del proceso.

## 8. Simulación

Un aspecto fundamental en el diseño de circuitos es tener la capacidad de simularlos para detectar errores en la descripción del comportamiento. Rara vez diseñamos un circuito bien a la primera. En muchas ocasiones es fácil detectar, pero no siempre es así (recuerda las veces que has visto que el diseño no hace lo que esperabas que hiciese y no has sabido por qué lo hace mal). Para solucionar los errores en la descripción, la simulación del circuito resulta de inestimable ayuda.

En la simulación se inserta el circuito a probar<sup>15</sup> en un banco de pruebas. El banco de pruebas es un modelo en VHDL que proporciona los estímulos al circuito para así poder ver si funciona como queremos. El modelo simula el funcionamiento real, y de ahí su nombre (simulación). La descripción del banco de pruebas suele incluir:

- Una referencia<sup>16</sup> ("*instanciación*" ) al circuito a probar
- Uno o varios procesos que generan los estímulos de entrada al circuito (también conocidos como vectores de test). Estos procesos simulan las entradas del circuito en funcionamiento real.
- Se puede incluir además procesos que comprueben automáticamente el correcto funcionamiento del circuito.
- Por último, la entidad del banco de pruebas no tiene entradas y salidas, ya que se generan todas los estímulos en procesos dentro de la arquitectura.

Los procesos del banco de pruebas no se van a implementar en hardware, son sólo para simulación. Por tanto, no se necesita que sean descripciones sintetizables, y en consecuencia, el tipo de descripción puede ser mucho más libre.

El esquema general de un banco de pruebas se muestra en la figura 8.1.

---

<sup>15</sup> Se suele abreviar con UUT (Unidad bajo pruebas: *Unit Under Test*)

<sup>16</sup> En inglés se dice *instantiation*, que algunos traducen como "*instanciación*" (que no existe en español) y otros como referencia (que sí existe)

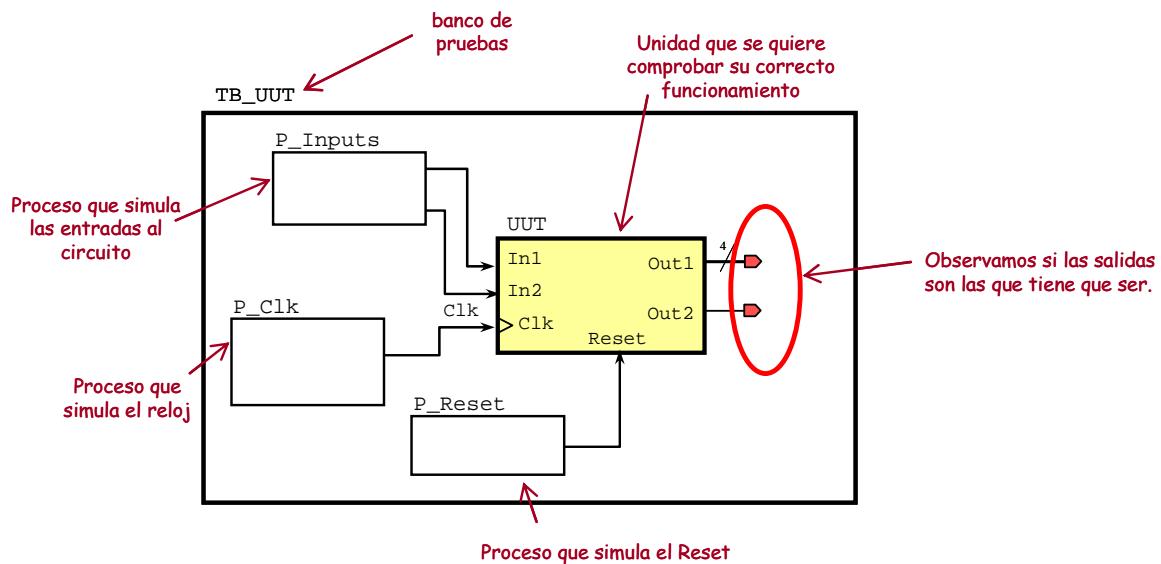


Figura 8.1: Esquema general de un banco de pruebas

A continuación vamos a ver los bancos de pruebas y la simulación con un ejemplo sencillo

### 8.1. Ejemplo sencillo

Vamos a realizar un contador muy básico que cuente de 0 a 15. Este contador va a contar ciclos de reloj, por tanto, si lo implementásemos en la placa no seríamos capaz de ver la cuenta por su rapidez.

Los puertos de entrada y salida del circuito se muestran en la figura 8.2. La salida del circuito (CUENTA) será un `unsigned` que contará de 0 a 15. Acuérdate de incluir la biblioteca `IEEE.NUMERIC_STD.ALL`, y quitar las dos últimas que se incluyen por defecto.

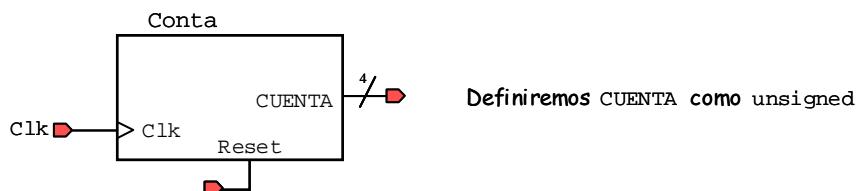


Figura 8.2: Puertos de entrada y salida del contador que vamos a probar

Así que crea un proyecto nuevo llamado `simul_conta` y crea el componente VHDL `Conta`. Este será un contador de 0 a 15 ascendente. Contará ciclos de reloj. Si no te acuerdas cómo se hacen los contadores, repasa el capítulo 6.

Una vez que lo tengas hecho y hayas comprobado que la sintaxis está bien, procederemos a simularlo. Para ello crearemos un banco de pruebas:

- Creamos una nueva fuente `Project → New Source`
- Indicamos que la fuente va a ser del tipo banco de pruebas de VHDL: selecciona `VHDL Test Bench` y nómbrala `TB_CONTA` (ver figura 8.3).

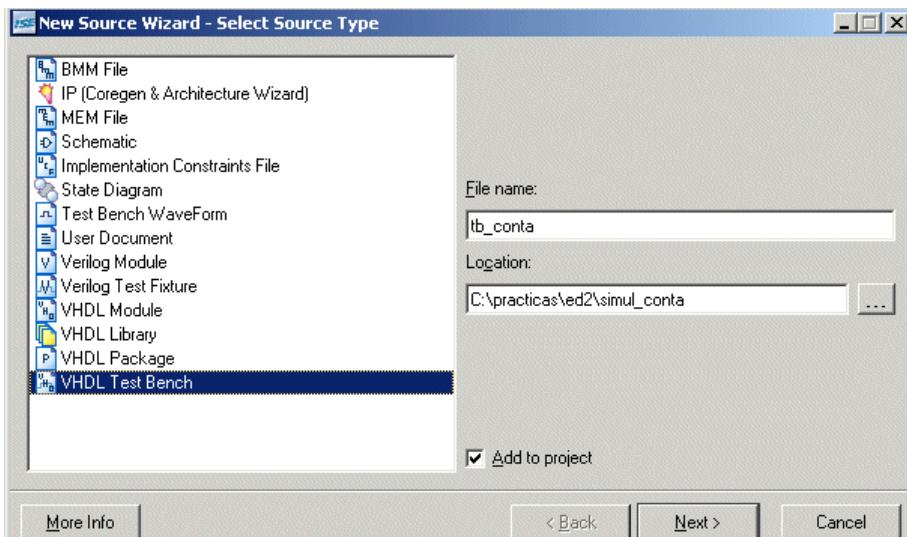


Figura 8.3: Selección de nueva fuente de tipo VHDL Test Bench

- En la siguiente ventana se selecciona la entidad que el banco de pruebas va a probar (*testear*). Como sólo hay una (*conta*), no hay más opción que seleccionar, por tanto pinchamos en *Next*. Y luego en *Finish*.

Como resultado nos aparecerá la entidad y la arquitectura VHDL del banco de pruebas. Este código se muestra en la figura 8.4. Este código tendremos que modificarlo, ya que el *ISE* crea un patrón general de banco de pruebas que tendremos que adaptar a nuestro circuito particular.

Para facilitar la identificación de las distintas partes del código, éstas se han señalado con letras de la A a la G (figura 8.4):

- Lo primero que tienes que hacer es poner la biblioteca `numeric_std` igual que hiciste con la entidad (figura 8.4-A). Recuerda el código 6-5.
- La entidad del banco de pruebas está la figura 8.4-B. Fíjate que esta entidad no tiene puertos.
- Dentro de la parte declarativa de la arquitectura (antes del `BEGIN` de la arquitectura) debes incluir:
  - La declaración del componente que vas a probar (figura 8.4-C). El nombre de los componentes y los puertos deben de coincidir con la declaración de la entidad (de `conta`, en este caso). Aquí tendrás que cambiar el tipo de la señal `cuenta`, pues si seguiste las instrucciones, debe ser un `unsigned`.
  - Las señales de entrada del componente, que irán a sus puertos: `reset` y `clk` (figura 8.4-D)
  - Las señales de salida del componentes, que irán a sus puertos: `cuenta`, que tendrás que cambiar a tipo `unsigned`. (figura 8.4-E)
- Ya en la parte de sentencias de la arquitectura (después del `begin`), viene la referencia ("*instanciación*") del componente (figura 8.4-F). Aquí se realiza el *mapeo* (*port map*) de los puertos del componente con las señales del banco de pruebas.
- Por último, el *ISE* ha creado un proceso vacío que tenemos que llenar para incluir los estímulos que le vamos a introducir a nuestro componente `conta10` (figura 8.4-G)

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
--USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY tb_conta_vhd IS
END tb_conta_vhd;

ARCHITECTURE behavior OF tb_conta_vhd IS

-- Component Declaration for the Unit Under Test (UUT)
COMPONENT conta10
PORT(
    Reset : IN std_logic;
    Clk : IN std_logic;
    cuenta : OUT std_logic_vector(3 downto 0)
);
END COMPONENT;

--Inputs
SIGNAL Reset : std_logic := '0';
SIGNAL Clk : std_logic := '0';

--Outputs
SIGNAL cuenta : std_logic_vector(3 downto 0);

BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: conta10 PORT MAP(
        Reset => Reset,
        Clk => Clk,
        cuenta => cuenta
    );

    tb : PROCESS
    BEGIN
        -- Wait 100 ns for global reset to finish
        wait for 100 ns;

        -- Place stimulus here
        wait; -- will wait forever
    END PROCESS;
END;

```

**A** Bibliotecas utilizadas  
Quitamos la std\_logic\_unsigned

**B** La entidad del banco de pruebas no tiene puertas de entrada/salida

**C** Declaración del componente que vamos a probar (UUT)

**D** Señales que vamos a generar (estímulos): entradas a conta,

**E** Señales que vamos a observar si están bien. Salidas del componente conta

**F** Referencia (instanciación) del componente conta (a probar)

**G** Patrón de ejemplo para generar los estímulos: Generar los valores de las señales de entrada, En este caso de Reset y Clk

Figura 8.4: Código creado automáticamente por el ISE para el banco de pruebas

Ahora realizamos los cambios indicados en la figura 8.4.

Antes de seguir con los cambios que tenemos que realizar, explicaremos la referencia al componente (figura 8.4-F). En la figura 8.5 se muestra las distintas partes de la referencia a un componente. La referencia a un componente es equivalente a insertar un componente en nuestra arquitectura. Al insertar el componente tenemos que indicar cómo se conectan los puertos ("port map"). Como un mismo componente se puede insertar varias veces (puede haber varias referencias a un componente), la referencia empieza con un nombre que la identifica de manera única. A continuación se incluye el nombre del componente que se referencia. Este nombre debe de coincidir con el nombre de la entidad de dicho componente.

Por último, en el mapeo de los puertos (*port map*) a la izquierda se ponen los nombres de los puertos del componente, y a la derecha los nombres de las señales de la arquitectura del banco de pruebas. Estos nombres, aunque los hemos puesto iguales, no tienen por qué coincidir. La figura 8.5 muestra de manera más gráfica esta explicación.

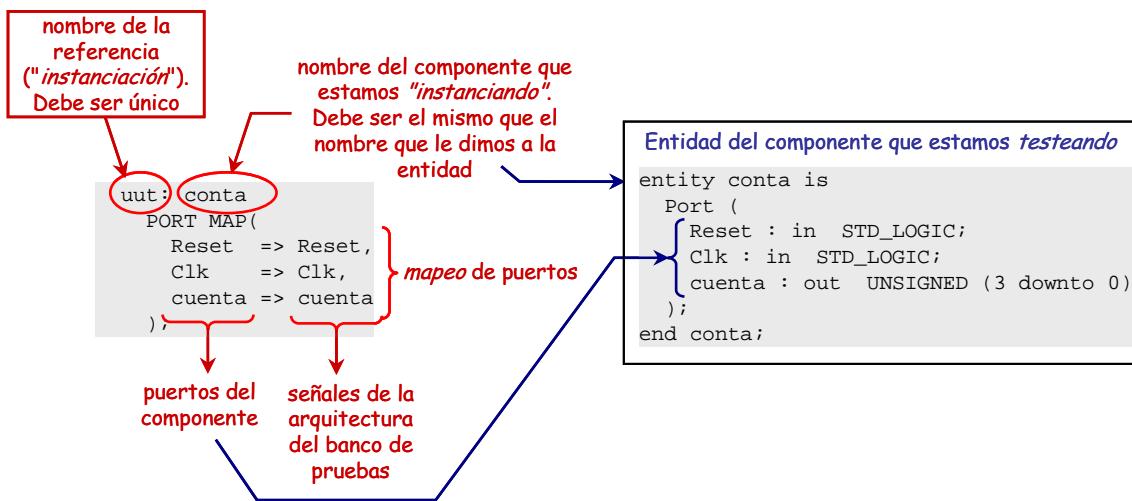


Figura 8.5: Distintas partes de la referencia ("instanciación") a un componente

Ahora sólo nos queda explicar cómo se generan los estímulos: el bloque G de la figura 8.4. En esta figura, el ISE incluye un proceso genérico que tenemos que rellenar con los estímulos que consideremos adecuados para comprobar nuestro circuito. Nosotros **borraremos** ese proceso para incluir los nuestros.

Tenemos dos entradas a nuestro circuito `Conta`: el reloj y el reset. Generaremos un proceso independiente para cada una de ellas:

El proceso del reloj debe de generar una señal de reloj periódica de 20 ns (50 MHz). Dicha señal de reloj estará la mitad del tiempo a cero y la otra mitad a uno (figura 8.6).

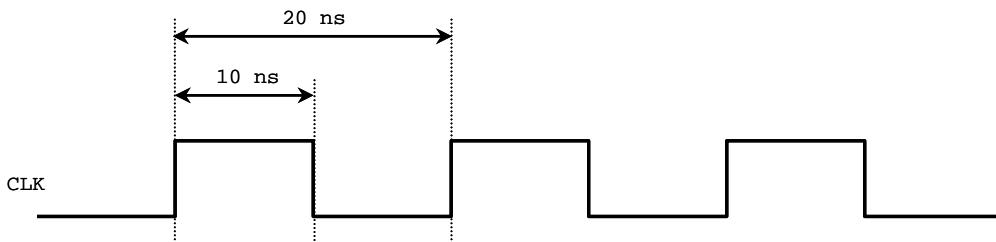


Figura 8.6: Periodo de la señal de reloj

Los procesos del VHDL son útiles para generar señales periódicas para simulación. Para ello se utilizan procesos sin lista de sensibilidad pero con sentencias "wait for". Al terminar un proceso, si no tiene lista de sensibilidad, el simulador vuelve a entrar inmediatamente en el proceso hasta encontrar una sentencia `wait`. La sentencia detiene la ejecución del proceso hasta que termine el tiempo indicado por la sentencia "wait for". Veámoslo con un ejemplo. Queremos realizar el proceso que genera la sentencia de reloj. Recuerda que estamos hablando para simulación, esta sentencia no se podrá implementar en la FPGA. El código 8-1 muestra cómo se genera una señal de reloj **para simulación**. Copia este proceso en el banco de pruebas.

```
P_Clk: Process
begin
  clk <= '1';
  wait for 10 ns;
  clk <= '0';
  wait for 10 ns; -- al terminar aqui, vuelve al principio
end process;
```

Código 8-1: Código que simula el funcionamiento del reloj de la placa Pegasus (50 MHz)

Ahora incluiremos la sentencia que simula la señal de reset. Esta señal no será periódica. En nuestro caso vamos a pulsar el reset una vez. La espera del wait la hacemos con un tiempo que no sea múltiplo del reloj para evitar que sean señales simultáneas, pues es algo improbable y puede llevarnos a alguna confusión en la simulación.

Para el reset no queremos que el proceso se vuelva a ejecutar una vez terminado, por lo tanto terminaremos el proceso con un `wait` que no tenga tiempo. Esto hará que el proceso se mantenga esperando eternamente (se acaba el proceso con ese `wait`).

```
P_Reset: Process
begin
    reset <= '0';
    wait for 45 ns; -- esperamos un tiempo no multiplo del reloj
    reset <= '1'; -- pulsamos el reset
    wait for 40 ns;
    reset <= '0'; -- soltamos el reset pasados 85 ns desde el principio
    wait; -- esperamos eternamente -> se para el proceso
end process;
```

Código 8-2: Código que simula el funcionamiento del reset

Incluye el proceso del código 8-2 en el banco de pruebas. Y ahora pasaremos a simularlo.

El primer paso es seleccionar el entorno del ISE para simularlo (figura 8.7).

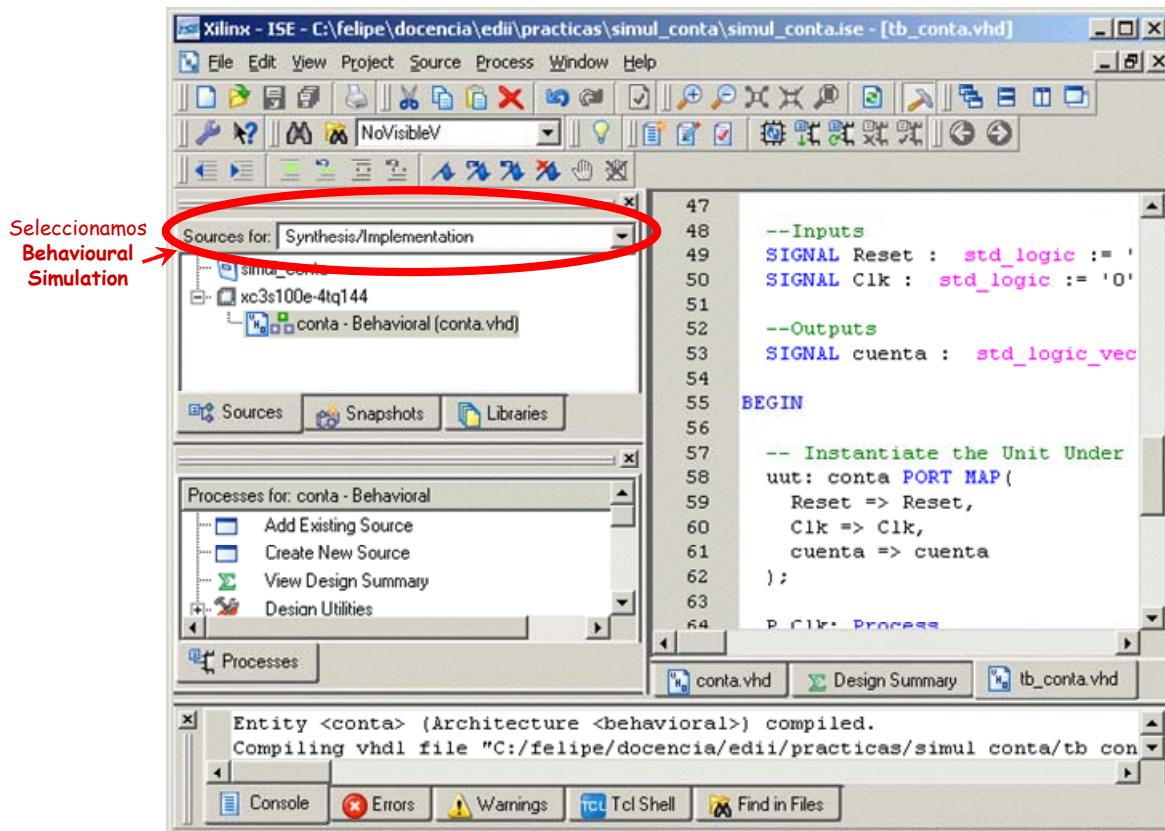


Figura 8.7: Selección del entorno para simulación

A continuación, en la ventana de procesos, te deberá aparecer Xilinx ISE Simulator (figura 8.8)

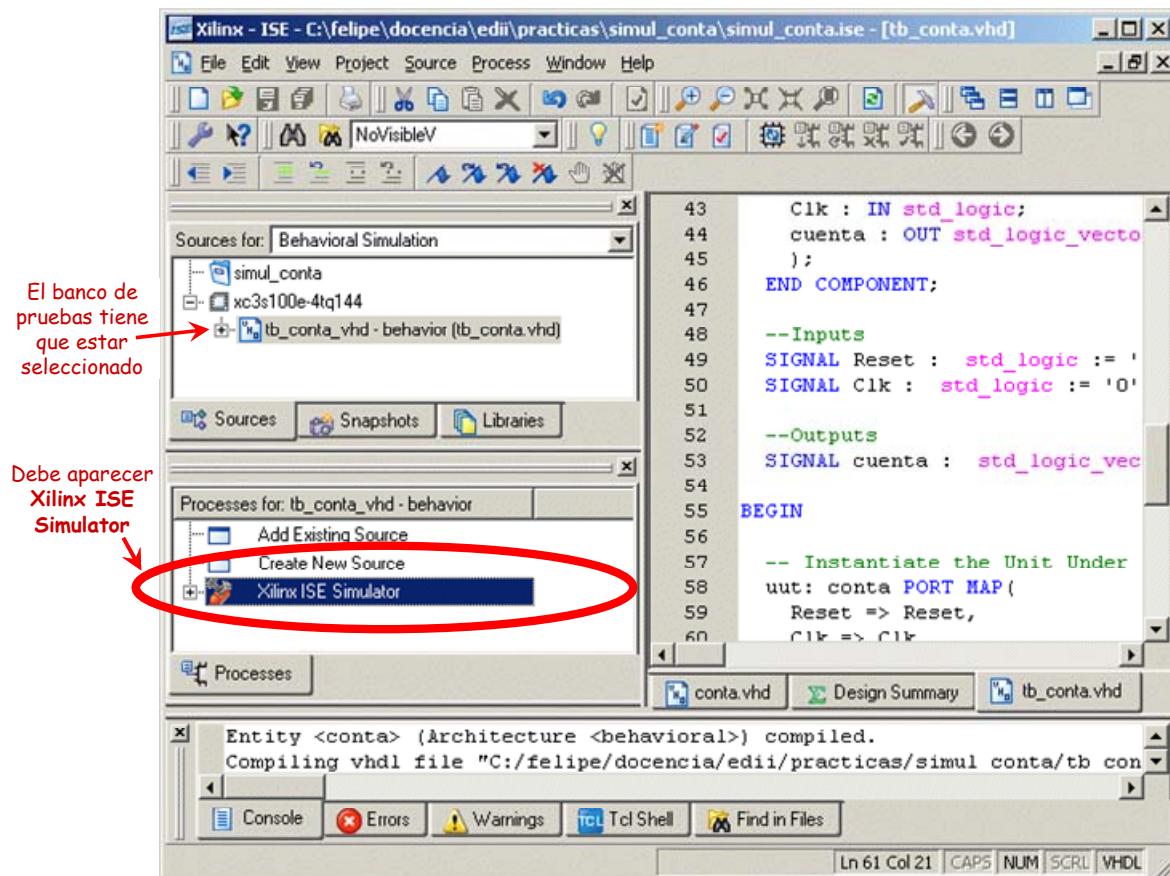


Figura 8.8: Entorno para simulación

Si en vez del *Xilinx ISE Simulator* te aparece *Modelsim* o cualquier otro, debes cambiar las propiedades del proyecto. Para ello, en la ventana de *Sources*, pincha con el botón derecho en el proyecto (donde pone algo parecido a *xc2s50-5pq208* si usas la *Pegasus* o *xc3s100e-4tq144* si es la *Basys*) y selecciona *Properties....* (figura 8.9).

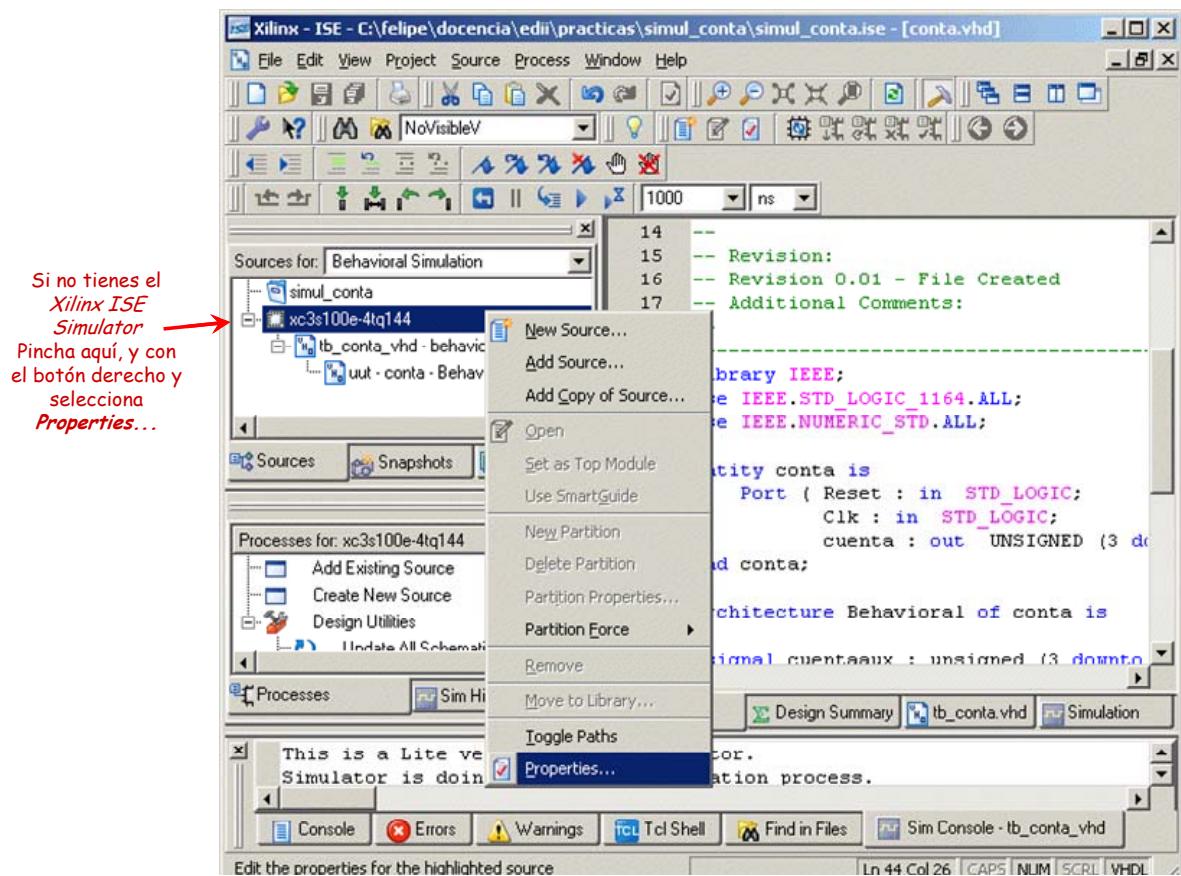


Figura 8.9: Cambio de las propiedades del proyecto

Al pinchar en *Properties*, te saldrá una ventana con la que podrás seleccionar el *Xilinx ISE Simulator* (figura 8.10). Luego pincha en *Ok*.

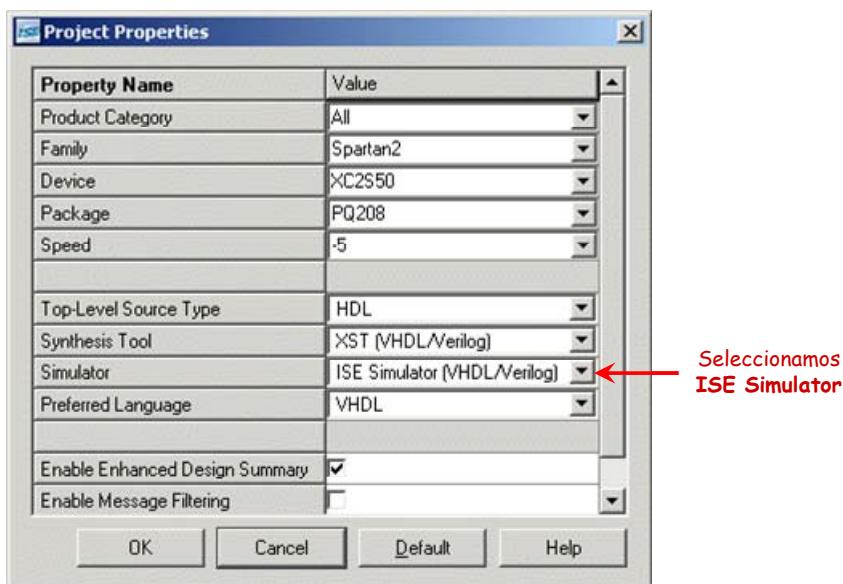


Figura 8.10: Selección del ISE Simulator

Ahora, en la ventana de *Sources* vuelve a seleccionar el banco de pruebas: *tb\_conta.vhd*. Y dentro de la ventana de *Processes*, pincha dos veces en *Simulate Behavioral Model* (figura 8.11).

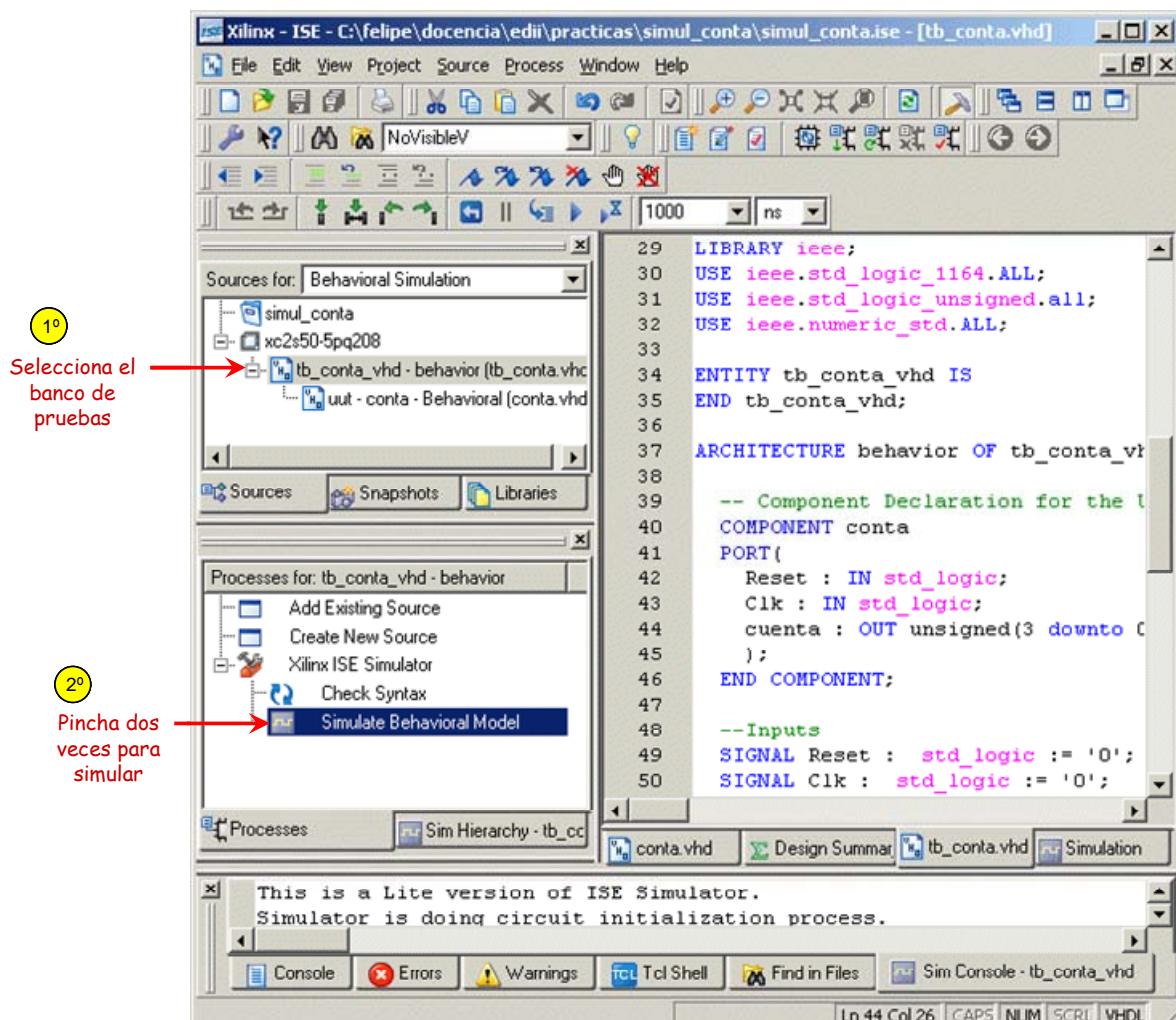


Figura 8.11: Arranque de la simulación

A continuación verás una ventana con el cronograma de la simulación (figura 8.12).

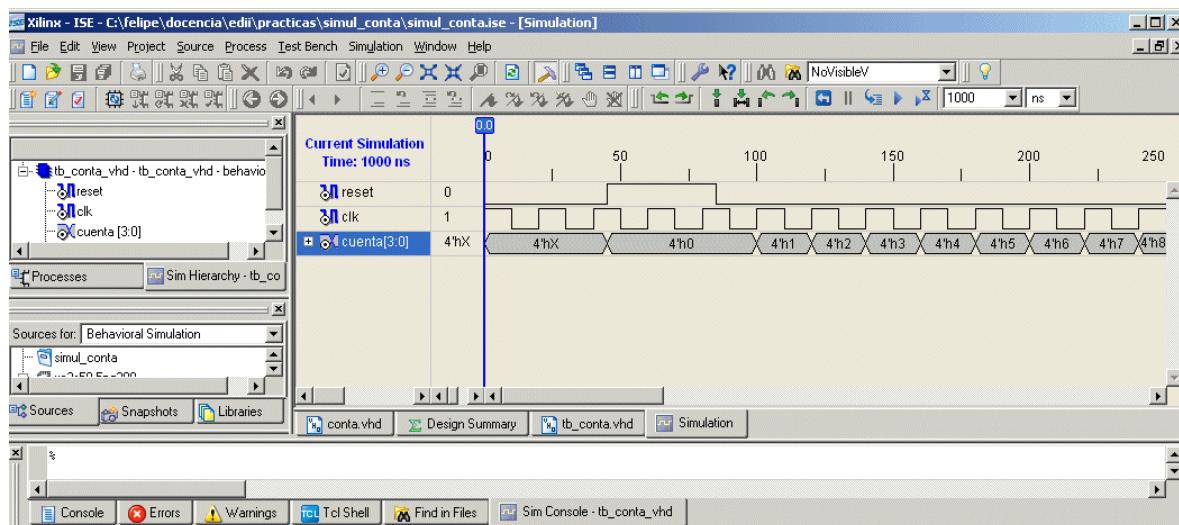


Figura 8.12: Resultado de la simulación

La señal `cuenta` no está en número decimal, para verlo más claro, pincha en ella con el botón derecho y selecciona *Decimal (unsigned)*. Mira la figura 8.13.

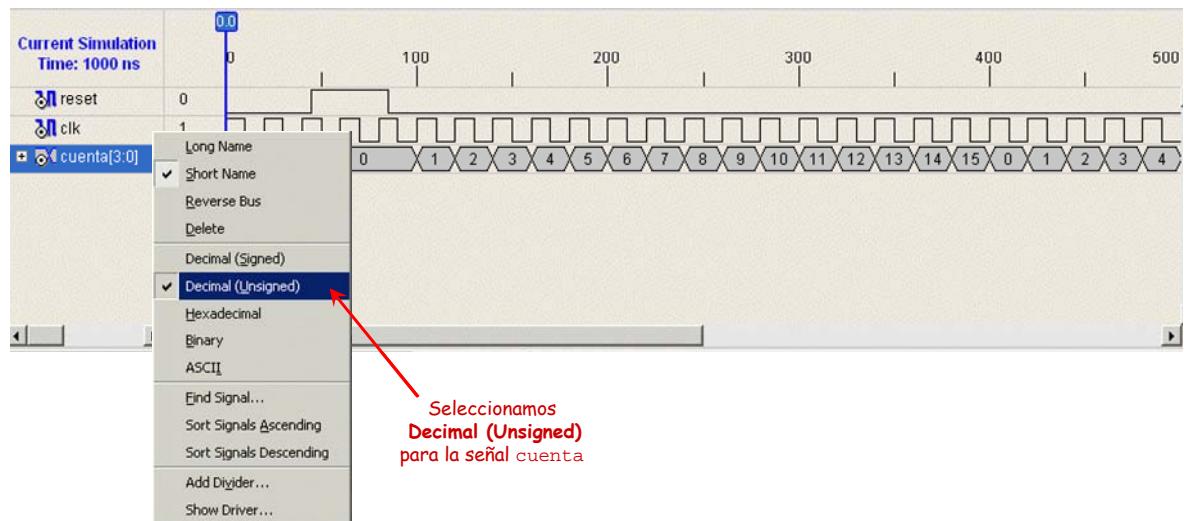


Figura 8.13: Visualización de cuenta como decimal sin signo

Ahora observa la simulación y comprueba que el contador realiza la cuenta correctamente. Fíjate que ocurre antes de dar el reset.

## 8.2. Ampliación

Ahora realiza la simulación de circuitos más complicados, que tengan más entradas que el reloj y el reset. Por ejemplo el contador manual del apartado 6.4 y los registros de desplazamiento. Para ello, tendrás que crear los procesos que generan dichas entradas.

## 8.3. Conclusiones

En este capítulo hemos visto cómo podemos comprobar que nuestro diseño funciona bien. Aunque a veces se pueden averiguar los errores con a partir de las salidas del circuitos, normalmente es resulta inmediato.

La comprobación de los diseños es una tarea que en muchos casos supera en tiempo de desarrollo a la propia tarea del diseño, por tanto, no se debe desestimar su importancia. En esos casos, la complejidad de los bancos de prueba puede ser muy elevada.

Así que en este capítulo sólo hemos hecho una primera aproximación a la simulación y comprobación de circuitos. En cursos posteriores veremos métodos más avanzados para la detección de errores [11].

## 9. Máquinas de estados finitos

Las máquinas de estados finitos las hemos estudiado en teoría, en esta práctica veremos cómo implementarlas en VHDL.

Aunque hay varias formas de implementar una máquina de estados en VHDL, de manera general, nosotros la implementaremos mediante tres procesos:

- Proceso secuencial que guarda el estado
- Proceso combinacional que obtiene el estado siguiente a partir del estado actual y de las entradas
- Proceso combinacional que obtiene las salidas a partir del estado actual y de las entradas en caso de que sea de Mealy.

Estos tres procesos se esquematizan en la figura 9.1. Fíjate que sólo el proceso secuencial tiene reloj y reset.

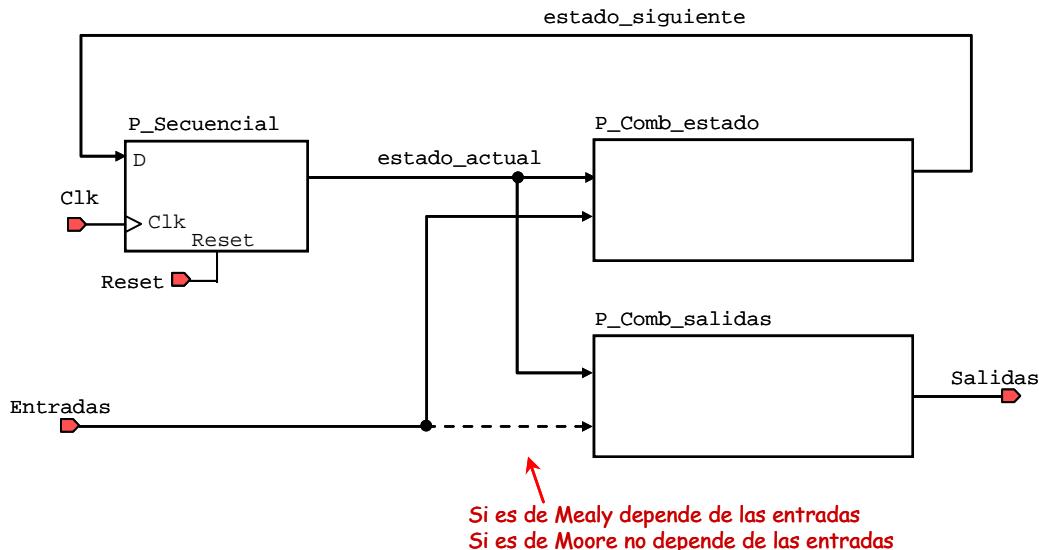


Figura 9.1: Esquema general de los procesos de una máquina de estados

Muchas veces se unen los procesos combinacionales en un sólo proceso. Sin embargo, te recomendamos que al menos al principio te acostumbres a hacerlo de esta forma más ordenada. También se puede hacer toda la máquina de estados en un único proceso.

Vamos a ver cómo se diseñan máquinas de estados finitos con un ejemplos.

### 9.1. Máquina de estados para encender y apagar un LED con pulsador

En realidad, aunque quizás no nos hayamos dado cuenta, ya hemos hecho algunas máquinas de estado finitos en algunos ejemplos. Sin embargo, como eran sencillas, pudimos diseñarlas de manera intuitiva. El circuito que enciende y apaga un LED con un pulsador que hicimos en la práctica 5.3 se puede hacer con máquinas de estados finitos.

Nuestro circuito tendrá dos bloques: el detector de flanco del pulsador (`BTN0`) y la máquina de estados finitos. El esquema de estos dos bloques se muestra en la figura 9.2 (estos bloques pueden tener más de un proceso).

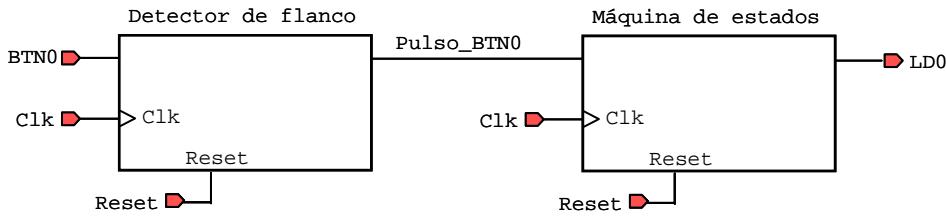


Figura 9.2: Bloques del circuito para encender y apagar un LED

Como el detector de flanco ya sabemos hacerlo (sección 5.3.2), sólo veremos la máquina de estados suponiendo que ya tenemos la señal acondicionada (`Pulso_BTN0`). Al implementarlo en la FPGA tendrás que crear el boque de detección de flancos.

El diagrama de transición de estados se muestra en la figura 9.3 (fíjate que está hecho como máquina de Moore). Cuando se presiona el pulsador se pasa de encendido a apagado, o de apagado a encendido. Recuerda de la sección 5.3.2 que necesitamos acondicionar la señal del pulsador para que se convierta en un pulso de un único ciclo de reloj. Por eso, en la leyenda del diagrama de la figura 9.3, la entrada es `Pulso_BTN0` y no directamente el pulsador: `BTN0`.

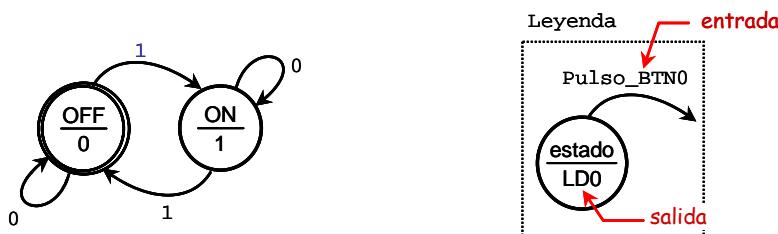


Figura 9.3: Diagrama de transición de estados para encender un LED con un pulsador

En las clases teóricas hemos aprendido a implementar el diagrama de la figura 9.3 con biestables y puertas lógicas. Ahora lo haremos en VHDL.

Lo primero que haremos es declarar un tipo enumerado que tendrá los estados de nuestra máquina. En nuestro caso serán `LED_ON` y `LED_OFF` (no pueden ser `ON` y `OFF` porque `ON` es una palabra reservada en VHDL).

```
type estados_led is (LED_OFF, LED_ON);
```

Código 9-1: Declaración de un tipo enumerado para la máquina de estados

La declaración del enumerado la pondremos en la parte declarativa de la arquitectura (antes del `BEGIN`). El tipo enumerado que hemos declarado en el código 9-1 nos permite asignar los valores `LED_OFF` y `LED_ON` a las señales que se declaren del tipo `estados_led`.

A continuación declararemos dos señales que indican el estado actual y el estado siguiente al que iremos según el estado actual y las entradas (mira la figura 9.1). Lo haremos así:

```
signal estado_actual, estado_siguiente : estados_led;
```

Código 9-2: Declaración de las señales de estado de tipo `estados_led` anteriormente declarado

Y ahora, en la parte de sentencias de la arquitectura, describiremos el diagrama de transición de estados en VHDL (figura 9.3). Según vimos en la figura 9.1 la máquina de estados se puede describir con tres procesos. Estos procesos se explican en los siguientes apartados.

### 9.1.1. Proceso combinacional que obtiene el estado siguiente

Este proceso obtiene el estado siguiente (el destino de la flecha en el diagrama de transición de estados) a partir del estado actual y las entradas. Este proceso es combinacional, por lo tanto no tiene reloj ni reset. Así que en la lista de sensibilidad tendremos todas las señales que se leen, que serán el estado actual y las entradas. En nuestro caso, observando el diagrama de transición de estados tendremos el siguiente proceso (figura 9.3):

```
P_COMB_ESTADO: Process (estado_actual, Pulso_BTN0)
begin
    case estado_actual is
        when LED_OFF =>
            if Pulso_BTN0 = '1' then
                estado_siguiente <= LED_ON;
            else
                estado_siguiente <= LED_OFF;
            end if;
        when LED_ON =>
            if Pulso_BTN0 = '1' then
                estado_siguiente <= LED_OFF;
            else
                estado_siguiente <= LED_ON;
            end if;
    end case;
end process;
```

Código 9-3: Proceso que obtiene el estado siguiente

Como vemos en el código 9-3, el proceso empieza con una sentencia CASE. Con esta sentencia seleccionamos el estado actual (estado\_actual) y según las entradas, obtendremos el estado siguiente (estado\_siguiente). Si lo comparas con el diagrama de la figura 9.3, ambos dicen lo mismo, uno en código y otro de forma gráfica.

Es muy importante poner todos los estados en el CASE. Y que en todas las alternativas se asigne el estado\_siguiente, porque de lo contrario, se generaría un *latch* para la señal estado\_siguiente. Y esta señal debe ser combinacional, ya que es en la señal estado\_actual donde se crea un elemento de memoria para guardar el valor.

### 9.1.2. Proceso secuencial

El proceso secuencial actualiza el estado actual en cada ciclo de reloj, y lo guarda en uno o varios elementos de memoria (*flip-flops*). En cada ciclo de reloj, el estado siguiente pasa a ser el estado actual. Para ello, simplemente usa biestables D. Por lo tanto es un proceso muy sencillo. El proceso incluye una sentencia de reset que indica el estado de partida.

```
P_SEQ_FSM: Process (reset, clk)
begin
    if reset = '1' then
        estado_actual <= LED_OFF;
    elsif clk'event and clk='1' then
        estado_actual <= estado_siguiente;
    end if;
end process;
```

Código 9-4: Proceso que actualiza y guarda el estado

### 9.1.3. Proceso combinacional que proporciona las salidas

Este proceso combinacional proporciona las salidas en función del estado actual y, en caso de que sea una máquina de Mealy, también en función de las entradas. En nuestro caso, como es una máquina de Moore, solo dependerá del estado actual. El proceso se muestra en el código 9-5

```

P_COM_SALIDAS: Process (estado_actual)
begin
    case estado_actual is
        when LED_OFF =>
            LD0 <= '0';
        when LED_ON =>
            LD0 <= '1';
    end case;
end process;

```

Código 9-5: Proceso que proporciona la salida

Recuerda que es muy importante que la salida esté asignada en todas las condiciones posibles para evitar generar *latches*.

Ya tienes los tres procesos de la máquina de estados. Ahora, realiza el diseño incluyendo también el detector de flanco. Implementa el circuito en la FPGA y comprueba que funciona bien.

## 9.2. Detector de flanco con máquinas de estados

Ahora vamos a implementar también el detector de flanco como máquina de estados. Es por tanto, una variante al que hicimos en el apartado 5.3.2, que lo hicimos mediante registros de desplazamiento. El detector de flanco de subida detecta una secuencia 01. Por tanto, es un ejercicio que sabemos hacer de clase. Como las entradas no están sincronizadas con el reloj de la placa (puedo pulsar cuando quiera), para evitar metaestabilidad, lo implementaremos como una máquina de Moore (recuerda figura 5.6) que son más seguras. En la figura 9.4 se muestra el diagrama de estados del detector de flanco de subida.

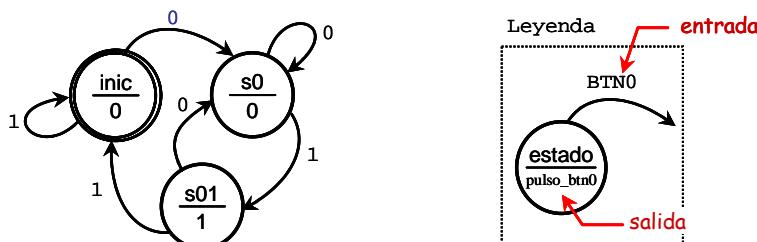


Figura 9.4: Diagrama de estados del detector de flancos (Moore)

Ahora crea un nuevo proyecto, importa los ficheros VHDL y UCF de la práctica anterior<sup>17</sup> y modifica el detector de flanco describiéndolo como la máquina de estados de la figura 9.4. Por tanto, creando otros tres procesos nuevos: uno secuencial, otro combinacional para el estado y el último combinacional para las salidas.

Tienes que tener en cuenta que vas a crear dos máquinas de estado en el mismo diseño, por lo tanto, tendrás dos tipos enumerados para especificar los estados: uno para el detector de flanco y otro para encender el LED. Por consiguiente, deben tener nombres diferentes. Por ejemplo, podrías declararlos como se muestra en el código 9-6

```

-- Estados para encender el LED
type estados_led is (LED_OFF, LED_ON);
signal estado_actual, estado_siguiente : estados_led;

-- Estados para detectar el flanco
type estados_flanco is (INIC, S0, S01);
signal est_fl_act, est_fl_sig : estados_flanco;

```

Código 9-6: Declaración de tipos enumerados distintos para cada máquina de estados

<sup>17</sup> Recuerda: Project → Add Copy of Source....

Implementa el circuito y comprueba que funciona igual que antes.

### 9.2.1. Variante

Si te fijas en la figura 9.4, una vez que se ha detectado el flanco, si viene un cero significa que de `BTN0` viene la secuencia 0101, y por lo tanto se generarán dos pulsos seguidos. Sin embargo es imposible que podamos pulsar dos veces en un intervalo de 40 ns. Por tanto, el nuevo pulso se debe a un rebote pulsador y en realidad es mejor no considerarlo.

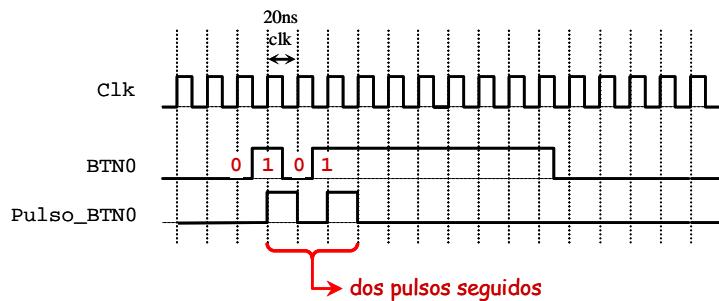


Figura 9.5: Cronograma de cuando llega la secuencia 0101

Así que es mejor realizar el diseño con el diagrama de estados de la figura 9.6, donde siempre que estamos en el estado `s01` volvemos al estado inicial. Por eso hay una `x`: en todos los casos, vuelvo a `INIC`.

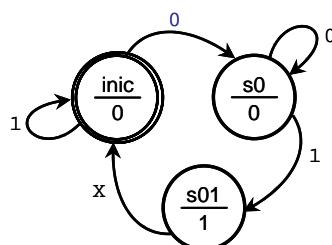


Figura 9.6: Diagrama de estados modificado

Realiza esta variante y comprueba que funciona igual.

### Comentario:

Los rebotes de los pulsadores es un aspecto muy importante en el diseño de circuitos electrónicos digitales. En la placa *Pegasus* están solucionados por medio de un circuito de condensadores y resistencias (figura 3.1). Sin embargo, la placa *Basys* no tiene este circuito (figura 3.2) y por tanto tenemos que solucionarlo mediante una máquina de estados y un temporizador de modo que no se consideren los pulsos de entrada si recientemente ya ha habido un pulso (suele ser durante unos 300 milisegundos). En el capítulo 11 aprenderemos a implementar este circuito, aunque estaría muy bien que fueses pensando en cómo lo harías. Mientras tanto, si usas la *Basys* tendrás que intentar presionar los pulsadores intentando que no haya rebotes (cogiéndole el truquillo algo se puede conseguir).

## 9.3. Desplazamiento alternativo de los LED

Para hacer esta práctica necesitas haber realizado la práctica 7.1 y las de este capítulo. Queremos iluminar los LED de la placa de la misma manera que lo hacía KITT (el Coche Fantástico) de modo que se empiece encendiendo el LED 0, luego el 1, y así sucesivamente hasta el 7, y al llegar al 7 vuelve al 6, 5, ..., hasta el 1 y el 0.

Hay muchas maneras posibles de plantear este problema. Nosotros los plantearemos de la siguiente manera:

- Usaremos un registro de desplazamiento que puede rotar a la izquierda y a la derecha según el valor de una señal `dsplza_izq`. Si vale '1' desplazará a la izquierda, si vale '0' a la derecha. Haremos rotación y no desplazamiento porque si no, en una vuelta, el registro se llenaría de unos (también se puede desplazar, pero asignando ceros al bit del extremo).
- El registro de desplazamiento tendrá una habilitación (`enable`). Si está a cero no habrá rotación en ningún sentido.
- El habilitación (`enable`) del registro de desplazamiento vendrá dada por un señal generada en un divisor de frecuencia. La señal tendrá una frecuencia de 1 décima de segundo. Una señal igual que la que generamos en el cronómetro (apartado 6.3, figura 6.5). Si quisiésemos que fuese más rápido o despacio, cambiaríamos la frecuencia de esa señal.
- El valor del registro pasa directamente a los LED.
- El registro de desplazamiento se resetea con un valor igual a "00000001". Esto hace que al resetear siempre se encienda el primer LED de la derecha: `LD0`.
- La orden de rotación a la izquierda o derecha (`dsplza_izq`) la produce una máquina de estados. Aparte del reloj y el reset, sus entradas son dos señales que indican si el LED que está encendido es uno de los extremos. Así, cuando el LED de la derecha esté encendido (`registro(0)`) se activará la señal `tope_dcha`, indicando que `LD0` está encendido. Si se enciende el LED de la izquierda, `registro(7)` valdrá '1' y por tanto se activará la señal `tope_izq`.
- Por tanto, la máquina de estados indicará el sentido de rotación de los LED hasta que llegue al tope de ese sentido y dará la orden de ir en sentido contrario hasta llegar al otro tope.

El esquema de este circuito se muestra en la figura 9.7, donde está el divisor de frecuencia, el registro de desplazamiento y la máquina de estados.

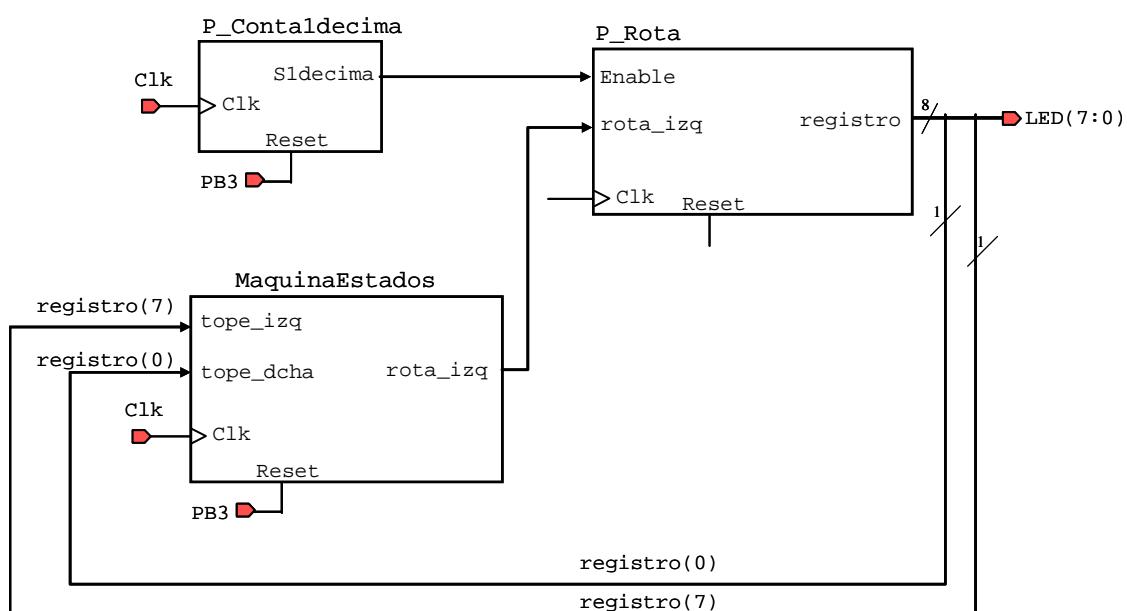


Figura 9.7: Esquema del circuito del movimiento alternativo de los LED

De los tres bloques de la figura 9.7, sólo la máquina de estados es un diseño totalmente nuevo. Pues el divisor de frecuencia se ha hecho en el apartado 6.3 (figura 6.5). Y el registro de desplazamiento es similar a los de la práctica 7.

A continuación se muestra el diagrama de estados de la máquina de estados, sin embargo, antes de verla, intenta deducirla por ti mismo. Pues en clase se han realizado problemas similares<sup>18</sup>.

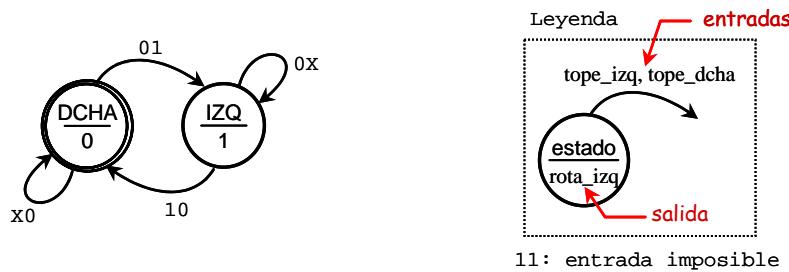


Figura 9.8: Diagrama de transición de estados

Puedes observar que se ha realizado como máquina de Moore.

Fíjate que hay entradas imposibles, y entradas que dan igual. En VHDL como la síntesis en biestables la va a realizar el *ISE-Webpack* y nuestra FPGA tiene tamaño de sobra para nuestro circuito, por ahora no nos vamos a preocupar de la simplificación. Por tanto, podemos diseñar el proceso combinacional del estado siguiente de la manera más cómoda para nosotros. Por ejemplo, como muestra el siguiente código, considerando sólo una señal de tope<sup>19</sup>:

```
P_COMB_ESTADO: Process (estado_act, tope_izq, tope_dcha)
begin
    case estado_act is
        when IZQ =>
            if tope_izq = '1' then
                estado_sig <= DCHA;
            else
                estado_sig <= IZQ;
            end if;
        when DCHA =>
            if tope_dcha = '1' then
                estado_sig <= IZQ;
            else
                estado_sig <= DCHA;
            end if;
    end case;
end process;
```

Código 9-7: Código del proceso combinacional que obtiene el estado siguiente

Para terminar, implementa el circuito entero en la FPGA y comprueba que funciona bien.

## 9.4. Conclusiones

Hemos visto cómo implementar máquinas de estados finitos en VHDL. Al enfrentarte con las especificaciones de un diseño, debes pensar cómo se transformarlas en una o varias máquinas de estados. Esta es una de las funciones más importantes del ingeniero. Posteriormente, el paso de la máquina de estados al VHDL es más o menos directo.

<sup>18</sup> El problema del carrito hecho en clase es similar, quizás un poco más complicado.

<http://laimbio08.escet.urjc.es/assets/files/docencia/EDII/carrito.pdf>

<sup>19</sup> Sin embargo, si lo hiciese a mano, sí que tengo que considerar las entradas imposibles para poder simplificar al máximo por Karnaugh



## 10. Clave electrónica

En esta práctica vamos a realizar un circuito que compruebe si se ha introducido una clave correctamente. Después de introducir la clave correcta, el circuito activará una salida, que en nuestro caso serán los LED, pero podría ser cualquier otra cosa, como abrir una puerta o desactivar una alarma.

En la placa *Pegasus* disponemos de cuatro pulsadores, que para simplificar los vamos a llamar **A**, **B**, **C** y **D**. Éstos se corresponderán con los pulsadores **BTN3**, **BTN2**, **BTN1** y **BTN0**. El reset irá en el primer interruptor (**SW0**). Las especificaciones del circuito son:

- La clave que activará la salida será **ACBA** (podría ser cualquier otra).
- Una vez que se ha introducido la clave correcta se encenderán los 8 LED durante 3 segundos. Tras lo cual se apagarán.
- Durante el tiempo en que los LED están encendidos (después de introducir la clave correcta), el circuito ignorará si se presionan los pulsadores.
- Despues de haber introducido la clave correcta y de que se hayan apagado los LED, se aceptará solapamiento para la siguiente clave. Esto es, la última **A** de la clave correcta podrá ser la primera **A** de la siguiente clave introducida (esto no tiene mucho sentido en una clave, pero se hace así como ejercicio).
- Se podrá tener un número infinito de intentos. La secuencia correcta se podrá introducir en cualquier momento excepto durante el tiempo en que los LED están encendidos.

Para este ejercicio sólo se darán unas indicaciones:

- Haz detectores de flanco para todos los pulsadores. Si usas la placa *Basys* tendrás más problemas para detectar una única pulsación y podría ser conveniente realizar un filtro temporal: no aceptar pulsos durante los 300 ms después de haber recibido un pulso (mira el capítulo 11).
- Haz el diagrama de estados a mano para el circuito detector de la clave
- Un esquema posible del circuito se muestra en la figura 10.1. Aparte del bloque detector de flancos, este esquema es muy similar a muchos problemas de examen, que tienen una máquina de estados y un circuito temporizador (ver capítulo 16 figura 16.1)
- Como la máquina de estados recibe entradas sincronizadas del mismo reloj: todas las entradas de este bloque provienen de bloques con el **clk** y no vienen del exterior; el circuito lo puedes hacer mediante una máquina de *Mealy* o de *Moore*.
- La señal **habilita\_cuenta** de la máquina de estados estará a uno hasta que se llegue al fin de la temporización (**fin\_cuenta='1'**). La señal **fin\_cuenta** valdrá '1' durante un sólo ciclo de reloj, al terminar la cuenta. Luego volverá a cero.
- Todo esto son recomendaciones, pero no tienes que hacer el circuito exactamente como se propone. Si tienes otra idea de cómo diseñarlo, llévala a cabo.

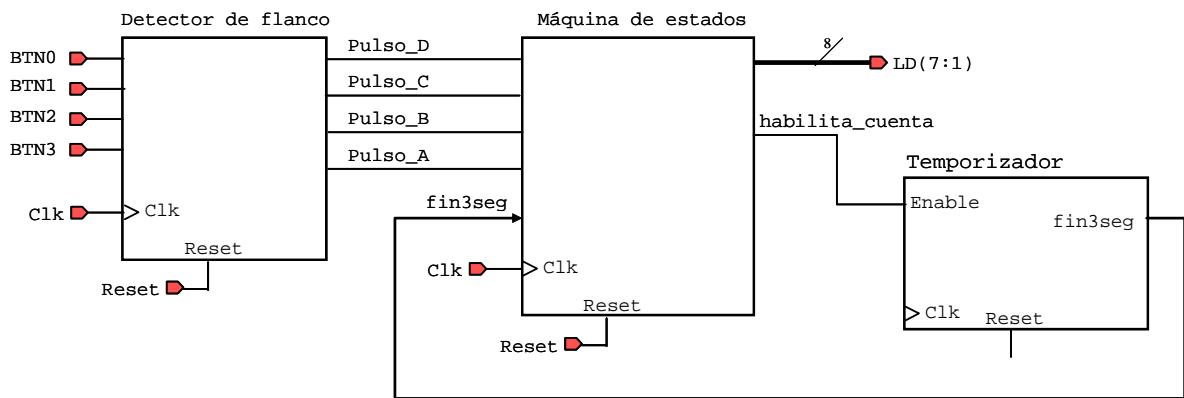


Figura 10.1: Esquema del circuito de la clave electrónica

### 10.1. Variantes

- Prueba a hacer la máquina de estados con *Mealy* y con *Moore*
- Haz que los LED parpadeen durante los tres segundos que se mantienen activos
- Haz el circuito sin solapamiento de la clave
- Muestra ordenadamente por los *displays* las letras que introduce el usuario: el último carácter introducido en el de la derecha, y el primer carácter (de los cuatro últimos) en el de la izquierda.

## 11. Circuito antirrebotes

En esta práctica se plantea el circuito antirrebotes que evita que el presionar un pulsador recibamos más de un pulso.

Este circuito es interesante para pulsadores que no tienen circuito antirrebotes en la propia placa. En la figura 11.1 se muestran los circuitos de los pulsadores de las placas *Pegasus* y *Basys*. A la izquierda se muestra el circuito que conecta el pulsador de la placa *Pegasus*. Este circuito tiene un condensador que hace filtrar los pulsos espurios que se producen por los rebotes al hacer contacto el pulsador. Sin embargo, la placa *Basys* no tiene estos condensadores, y por tanto, se producen más transiciones espurias. Estas transiciones pueden provocar que al pulsar una vez, el circuito interprete que se ha pulsado varias veces. Así que tenemos que realizar un circuito antirrebotes en VHDL.

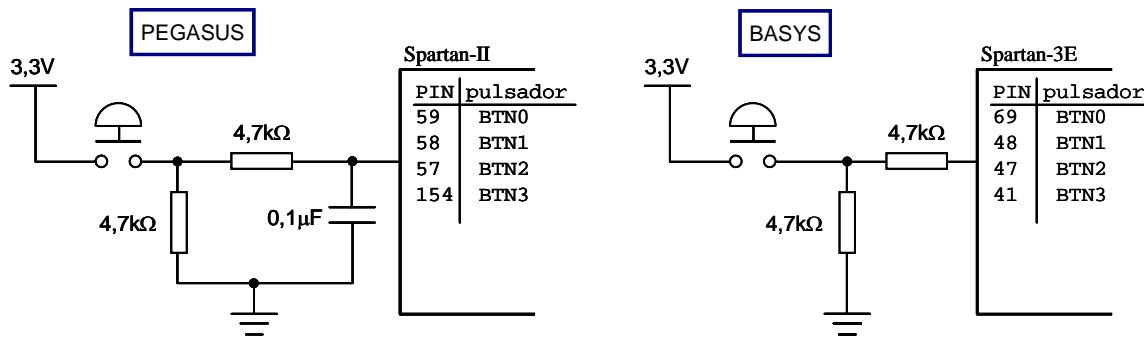


Figura 11.1: Esquema de los pulsadores en la placa Pegasus y en la placa Basys

La figura 11.2 muestra el cronograma resultante del detector de flancos (`PULSO_BTN`) cuando hay rebotes en la entrada (`BTN`). Como para este ejemplo existen 4 flancos de subida, el detector de flancos producirá cuatro pulsos. Sin embargo, sólo queremos que se produzca uno.

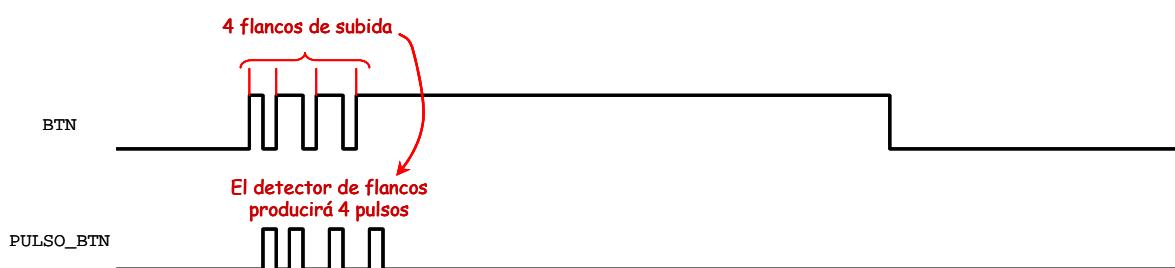


Figura 11.2: Cronograma de la salida del detector de flancos (`PULSO_BTN`) cuando hay rebotes en la entrada (`BTN`)

Para evitar los rebotes basta con eliminar todos los pulsos que se generen durante los siguientes 300 milisegundos después de haber detectado un flanco de subida. Si conseguimos hacer este circuito, la salida filtrada (`FILTRO_BTN`) sería como la mostrada en la figura 11.3.

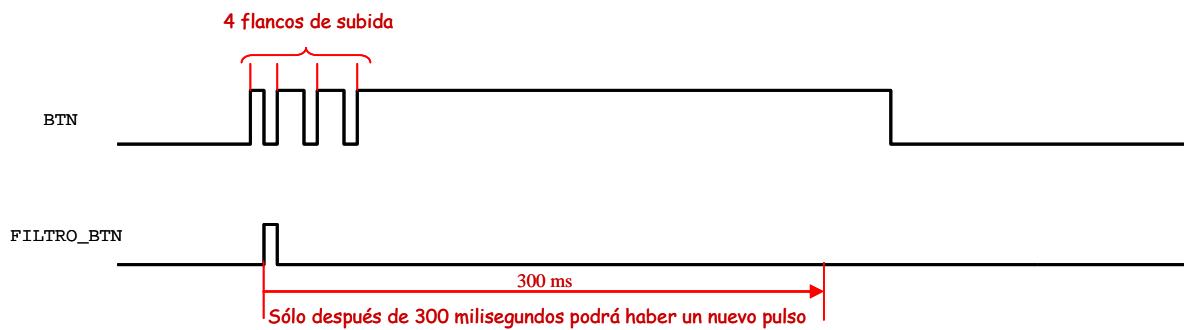


Figura 11.3: Cronograma de la salida del filtro antirrebotes (**FILTRO\_BTN**) cuando hay rebotes en la entrada (**BTN**)

¿Cómo podríamos realizar este circuito? pues con una máquina de estados similar al detector de flancos pero con temporizador. El esquema del circuito se muestra en la figura 11.4. En este esquema propuesto, el temporizador se habilita cuando se ha detectado un flanco de subida. A partir de entonces no se producirá ningún pulso de subida hasta que el temporizador dé la señal de que han pasado 3 milisegundos. Después de esto, se podrá detectar un nuevo flanco de subida.

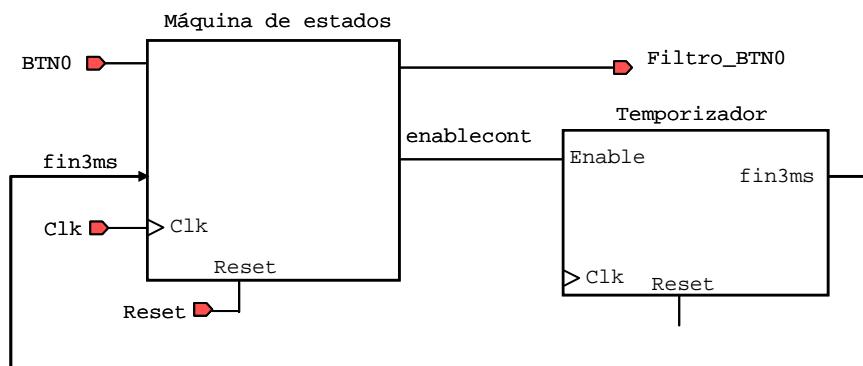


Figura 11.4: Esquema del circuito que filtra los rebotes de la entrada

Intenta pensar por ti mismo el diagrama de estados. Si no se te ocurre, puedes mirar la figura 11.5.

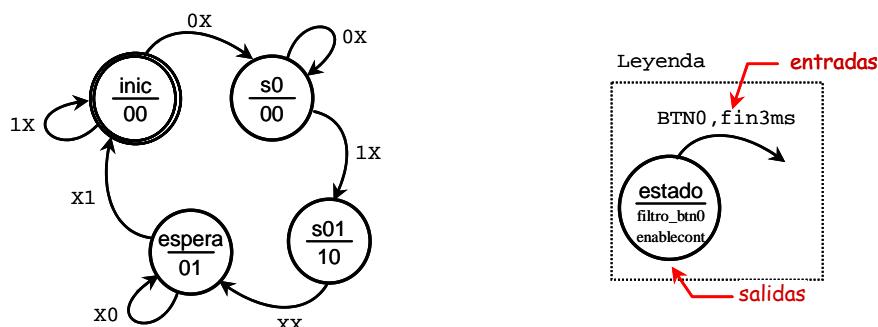


Figura 11.5: Diagrama de estados del circuito que filtra los pulsos

Para cada pulsador habría que realizar un circuito similar.

Implementa el contador manual de la sección 6.4 con estos filtros. Utiliza una placa *Basys* para comprobar que realmente funciona bien.

## 12. Máquina expendedora

En esta práctica realizaremos distintas variantes de una máquina expendedora de bebidas. Empezaremos por una versión sencilla.

### 12.1. Versión sencilla

La primera versión de esta máquina expendedora sólo tiene un tipo de bebida que cuesta 1,5 euros. Sólo acepta cuatro tipos de moneda: 1 euro, 50 céntimos, 20 céntimos y 10 céntimos. Y lo peor: ¡no devuelve cambio!

La máquina mostrará por un *display* el precio de la bebida cuando no se haya introducido ninguna moneda y, después de haber introducido monedas, el *display* mostrará la cantidad de dinero que se ha introducido. Esta primera versión de la máquina sólo podrá ser utilizable por estudiantes de electrónica digital, ya que mostraremos los valores por un único *display* codificado en hexadecimal. Esto es, si queremos indicar 1,5 € mostraremos el número 15 en hexadecimal: F. Si queremos mostrar 1,4 €, mostraremos 14 en hexadecimal: E. Y así sucesivamente, por ejemplo, para indicar 1 euro: A. Para indicar 50 céntimos (0,5 €) mostraremos 5. Con esto conseguimos una versión más sencilla, ya que utilizamos un sólo *display*. Esta versión se podrá ir mejorando en las siguientes prácticas.

Después de haber introducido 1,50 euros (o más, pero sin dar la vuelta), la máquina activará la salida para que abra la compuerta que deje salir la lata de bebidas. Nosotros, como no tenemos compuerta, encenderemos los LED durante 2 segundos para indicar al usuario que ha introducido la cantidad suficiente de dinero. Opcionalmente en el *display* se mostrará la letra L (de lata). Durante este tiempo no se aceptan monedas.

Como tampoco tenemos sensores para la entrada de monedas, utilizaremos los cuatro pulsadores para indicar que han entrado monedas. BTN3 será el indicador de que entra una moneda de 1 euro, BTN2 de 50 céntimos, BTN1 de 20 céntimos y BTN0 de 10 céntimos.

El esquema de entradas y salidas de nuestro circuito de control de la máquina expendedora será como el mostrado en la figura 12.1. A la derecha se muestra el significado de los puertos de entradas.

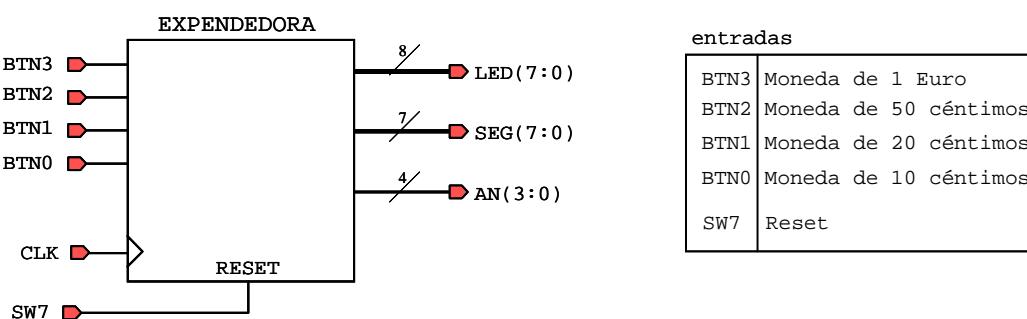


Figura 12.1: Esquema de puertos de entrada y salida del control de la máquina expendedora

¿Cómo haríamos este circuito por dentro? de las prácticas anteriores ya puedes tener una idea. El esquema es similar al de la clave electrónica (recuerda la figura 10.1). La figura 12.2 muestra un esquema de este circuito. Como puedes ver, incluye:

- Un bloque de detección de flanco para los pulsadores, haciendo que sólo se considere una moneda cada vez que se presiona un pulsador.
- El bloque de la máquina de estados, que debe saber cuánto dinero se ha introducido y controlar las salidas.
- El bloque de temporización, que debe indicar el paso de 2 segundos después de haber introducido el dinero suficiente para dar una lata.
- El bloque que convierte a siete segmentos la cantidad de dinero (en este caso un número hexadecimal de una cifra: 4 bits) que se quiere mostrar por el *display*. Si quisiésemos mostrar una L cuando durante la temporización, tendríamos que incluir la señal *enable\_cont* como entrada del temporizador.

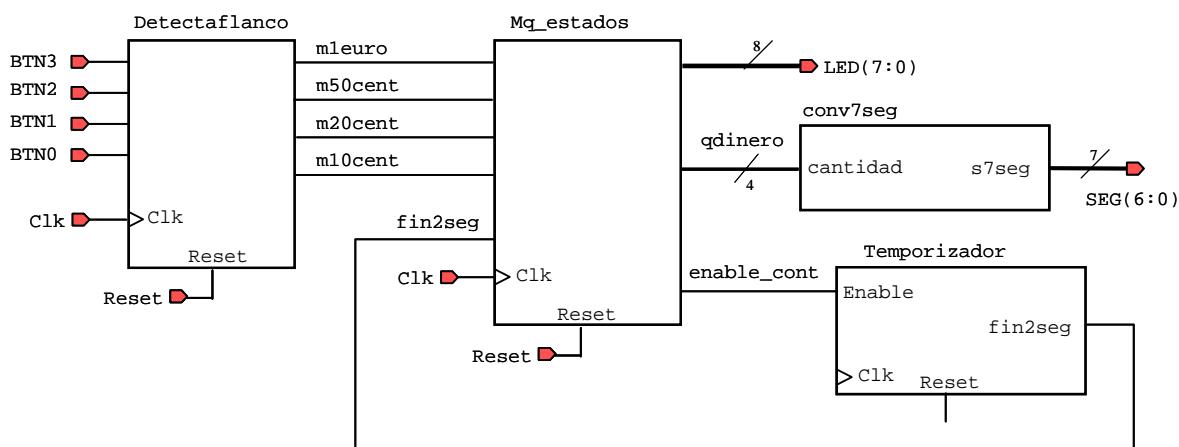


Figura 12.2: Esquema interno del circuito de control de la máquina expendedora simple

Haz el diagrama de estados de la máquina de estados, implementa el circuito y comprueba que funciona bien.

## 12.2. Versión decimal

El circuito que hemos hecho está bien para estudiantes de electrónica digital, pero si lo quiere utilizar alguien que no sabe hexadecimal se va a hacer un lío.

Así pues, vamos a ser más explícitos con los mensajes que vamos a dar.

EL diseño va a ser exactamente igual con la diferencia de la información que se muestra en los *displays*. Ahora utilizaremos los cuatro *displays* (recuerda la práctica 6.3.2).

Lo que se va a mostrar según el estado se muestra en la figura 12.3.

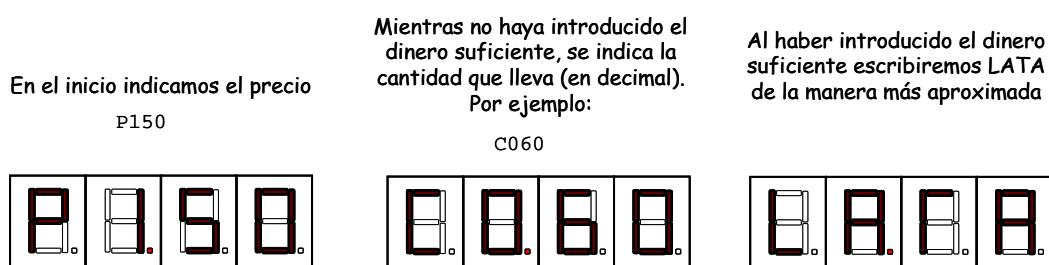


Figura 12.3: Indicaciones de los displays según el estado

## 13. Control con PWM

La modulación por ancho de pulso (*pulse width modulation*: PWM) es un tipo de control que se puede utilizar para el control de motores eléctricos de continua. En esta práctica lo utilizaremos para controlar la intensidad de un LED, pero con un circuito de potencia adecuado podrías controlar un motor de corriente continua.

### 13.1. Funcionamiento del PWM

Explicaremos de manera simplificada el funcionamiento de un PWM, este se ve con más detalle en la asignatura de Sistemas Electrónicos Digitales.

Los parámetros fundamentales del PWM son el periodo ( $T$ ) y el ciclo de trabajo ( $D$ ). El ciclo de trabajo indica el tiempo que la función vale uno respecto al tiempo total (el periodo). La figura 13.1 muestra tres ciclos de trabajo distintos. Observa que el periodo del PWM se mantiene constante, y lo que cambia es el tiempo en que la señal se mantiene a uno respecto al periodo total.

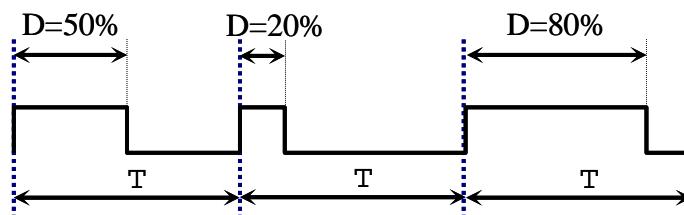


Figura 13.1: Señal PWM que se le ha cambiado el ciclo de trabajo

Si el periodo del PWM es suficientemente pequeño, el dispositivo que está gobernado (por ejemplo, el motor) no notará las variaciones de la tensión y el resultado es que el motor recibirá una corriente promedio dada por el ciclo de trabajo. La figura 13.2 muestra dos PWM con ciclos de trabajo distintos.

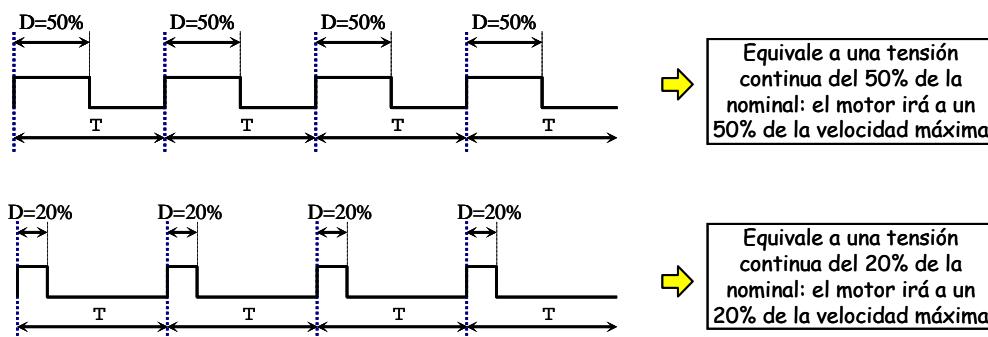


Figura 13.2: Control de velocidad con el PWM

### 13.2. Control de la intensidad de un LED

Sabiendo el funcionamiento básico del PWM, lo que vamos a hacer es controlar la intensidad de un LED mediante un PWM. El control lo haremos con dos pulsadores. La intensidad irá desde cero hasta 7. Esto es, cuando valga cero, los LED no lucirán, cuando valga 7, los LED lucirán en su máxima amplitud.

En los sistemas digitales los cambios en el ciclo de trabajo son discretos. Es decir, no se puede poner cualquier ciclo de trabajo, depende del número de bits que utilice.

Por ejemplo, para nuestro enunciado, si utilizo 3 bits para el PWM, puedo contar de 0 a 7, y puedo utilizar un PWM con los ciclos de trabajo mostrados en la figura 13.3.

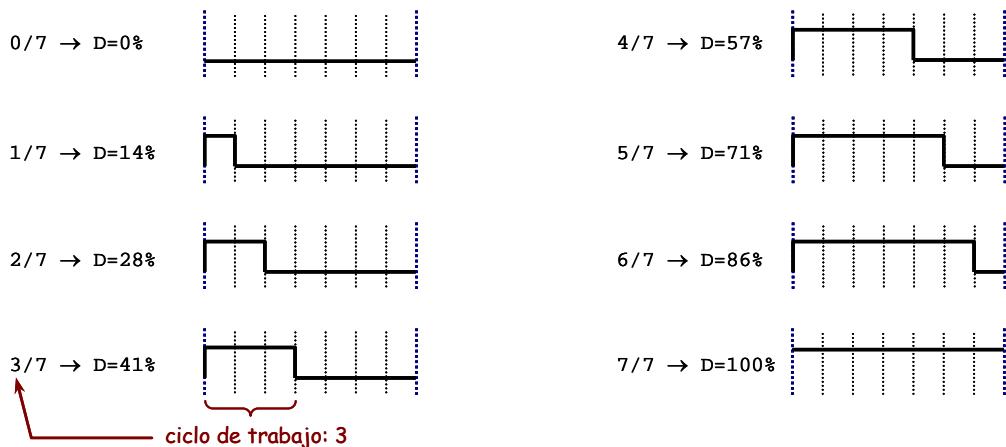


Figura 13.3: Los 8 ciclos de trabajo posibles con un PWM de 3 bits de resolución

Para la realización de la práctica necesitamos:

- Generar el ciclo de trabajo (*dutycycle*): lo haremos con un contador igual al de la práctica 6.4, y de la misma manera mostraremos el ciclo de trabajo por el *display*.
- Generar la señal periódica que produzca una señal con periodo T (*cuentaPWM*). En realidad será un contador. ¿De cuánto sería este contador? No es tan directo saberlo, te recomiendo dibujar el cronograma o simularlo.
- Comparar la cuenta de la señal periódica con el valor del ciclo de trabajo, si el ciclo de trabajo es mayor que la señal del PWM será uno, y si no, será cero.

### 13.3. Ampliación a control de motores

Mediante un circuito de potencia adecuado podríamos controlar un motor. Habitualmente se usa un puente en H. El puente en H (*H-bridge*) lo podemos hacer nosotros, o bien comprar un módulo PMOD de *digilent* (*PMODHB3*<sup>20</sup> o *PMODHB5*<sup>21</sup>). Además, el puente en H nos permitiría controlar el sentido de giro.

En la práctica 14 se incluye la descripción detallada de un circuito para controlar un motor paso a paso, que se podría adaptar para este caso.

Pero de manera más sencilla, se puede utilizar un transistor de potencia, como se hace en las prácticas de la asignatura Sistemas Electrónicos Digitales del próximo curso. El esquema básico del circuito se muestra en la figura 13.4. Según el motor, se escogerá la tensión (*V<sub>cc</sub>*), el transistor y resistencias adecuadas.

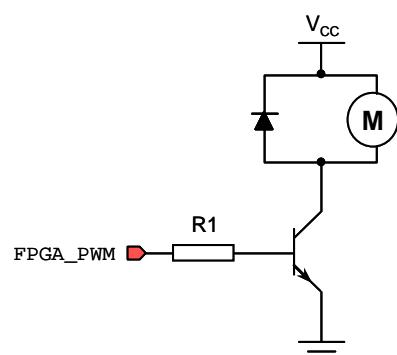


Figura 13.4: Circuito de potencia para controlar el motor desde la FPGA

<sup>20</sup> PMODHB3: <http://www.digilentinc.com/Products/Detail.cfm?&Prod=PMOD-HB3>

<sup>21</sup> PMODHB5: <http://www.digilentinc.com/Products/Detail.cfm?&Prod=PMOD-HB5>

## Circuitos digitales y analógicos

A continuación veremos unas prácticas en las que además de la parte relativa a la FPGA y el VHDL, realizaremos los circuitos de potencia y analógicos necesarios para interactuar con el mundo real. Los conocimientos de electrónica analógica no se explicarán en detalle, ya que se corresponden con la asignatura de Electrónica Analógica.



## 14. Control de motor paso a paso

En esta práctica realizaremos el circuito de control de un motor paso a paso (*stepper motor*). Con lo que ya sabemos, el control digital de este tipo de motores no debería costarnos mucho. Lo más importante de esta práctica es que se incluirá el diseño de la electrónica de potencia, que permitirá manejar el motor mediante la FPGA.

Primero veremos cómo funcionan los motores paso a paso, luego cómo podemos implementar la parte de control digital (el VHDL), por último veremos el circuito de potencia que adapta las salidas de la FPGA para manejar el motor.

---

### 14.1. Motores paso a paso

Hay mucha información sobre los motores paso a paso en internet<sup>22</sup>, por ello no nos extenderemos en cómo están hechos y su funcionamiento. Aquí simplemente se incluirá la información necesaria para poder realizar en circuito de control.

Sin entrar en las características y en cómo están fabricados los motores (imanes permanentes, reluctancia variable o híbridos), cuando tenemos un motor paso a paso, tenemos que distinguir si es bipolar o unipolar.

Normalmente los unipolares tienen 5 cables, y los bipolares 4 cables. Algunos unipolares tienen 6 cables, habiendo dos cables que son el de alimentación.

Aquí vamos a explicar el unipolar, ya que muchas de las impresoras antiguas tienen algún motor paso a paso unipolar, y así no tendremos que comprar ningún motor para hacer esta práctica. Hemos desguazado varias impresoras y escáneres, y suelen tener algún motor paso a paso (sobre todo las antiguas), también suele haber motores de continua, así que tendrás que distinguirlos. Muchas veces los motores tienen una etiqueta que indica que es un motor paso a paso (*stepping motor*) y en otros tienes que mirar su referencia en la página web del fabricante. Nosotros hemos encontrado motores del fabricante Mitsumi [17] y Minebea [16]. En las páginas web de estos fabricantes se pueden encontrar las características de los motores.

#### 14.1.1. Identificación de terminales

Una vez que hemos conseguido un motor unipolar, tenemos que encontrar los devanados opuestos y el terminal de alimentación. El procedimiento para descubrir qué terminal es cada uno es el siguiente<sup>23</sup>:

##### 1. Averiguar cuál es el terminal de alimentación

Según la figura 14.1, detectar el terminal de alimentación es fácil, pues si medimos las resistencias con un polímetro, la resistencia del terminal de alimentación respecto a cualquier otro terminal es la mitad que la resistencia que hay entre dos terminales cualesquiera que no sean el de alimentación. En caso de que haya 6 terminales, posiblemente dos de ellos correspondan al de alimentación y por tanto la resistencia entre ellos será cero.

---

<sup>22</sup> Información de los motores paso a paso en español: [7, 24] y en inglés [4, 9, 21, 22]

<sup>23</sup> Información obtenida de <http://www.doc.ic.ac.uk/~ih/doc/stepper/others/>

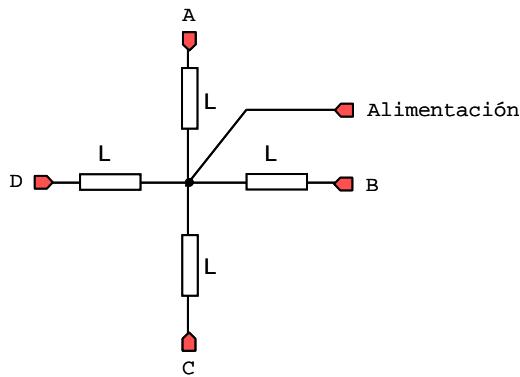


Figura 14.1: Conexión interna de un motor unipolar

## 2. Identificar los terminales opuestos

Primero conectamos el terminal de alimentación a la fuente ( $V_{cc}$ ) y ponemos cualquier otro terminal a tierra. A este terminal que conectamos a tierra le llamaremos terminal A. En la figura 14.2 se representa esta conexión. Al hacer esto vemos que el motor se posiciona y que cuesta más moverlo.

A continuación, manteniendo el terminal A a tierra y el terminal de alimentación conectado a la fuente, vamos probando qué sucede al conectar cada uno de los demás terminales a tierra. Esto lo hacemos por separado, es decir, en cada momento hay como máximo dos terminales a tierra.

Aquel terminal que no haga girar un paso al motor sino que haga que pierda par resistivo será el terminal opuesto a A, esto es, será el terminal c (según la referencia de la figura 14.2). En los otros dos casos, habrá un pequeño giro del motor en sentidos opuestos. En la figura 14.3 se muestra este concepto de manera gráfica.

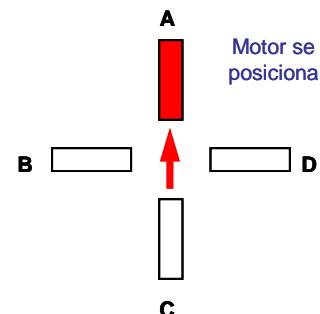


Figura 14.2: El terminal A a tierra

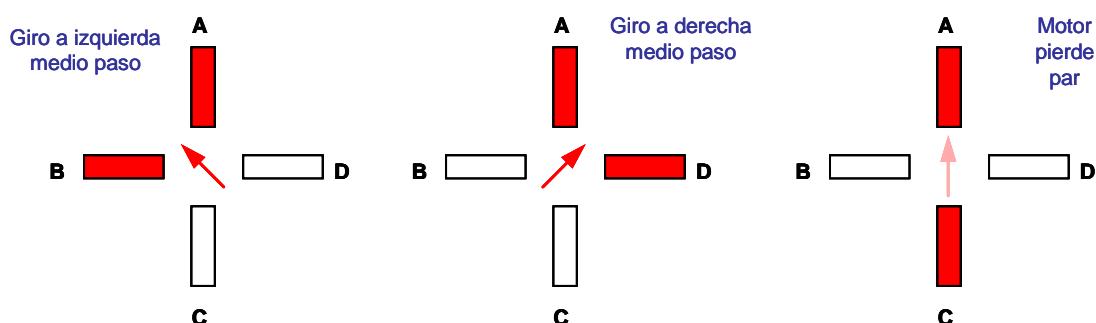


Figura 14.3: Resultado de poner a tierra cada uno de los otros tres terminales, cuando el motor pierde par indica que es el correspondiente

Una vez que tengamos los terminales identificados, ya podemos generar las secuencias de giro. En la figura 14.4 se muestra la identificación de los terminales para el motor que hemos obtenido de impresoras Hewlett Packard modelos DJ560C y DJ690c. Este motor es el modelo PM55L-048 de Minebea, cuyas hojas de características se pueden consultar en la página web de Minebea [16]. De todos modos, el funcionamiento con otros motores será similar, aunque tendrás que leer las características del motor que tengas para adaptar el circuito.

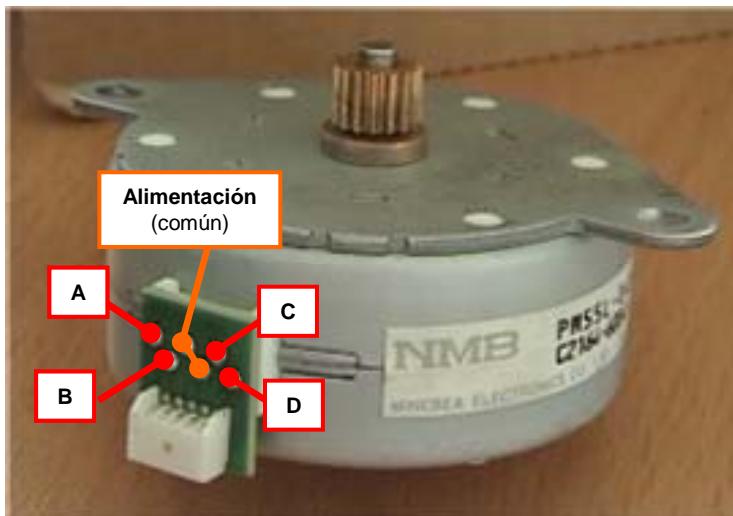


Figura 14.4: Identificación de los terminales en nuestro motor PM55L-048 de Minebea

#### 14.1.2. Secuencia del motor paso a paso

Para hacer girar al motor tenemos que ir poniendo a tierra algunos terminales en una secuencia concreta. Poner a tierra un terminal hace que circule corriente por el devanado correspondiente, y esto hace que se mueva el rotor del motor. De la figura 14.3 podemos intuir cómo podemos obtener la secuencia con la que podemos hacer que el motor gire. La figura 14.5 aclara cómo obtener esa secuencia. La flecha del centro indica de manera esquemática la posición del rotor. De todos modos, es sólo un esquema, no se producen giros de 90° con cada paso, sino que son giros mucho menores (suelen ser menores de 10°).

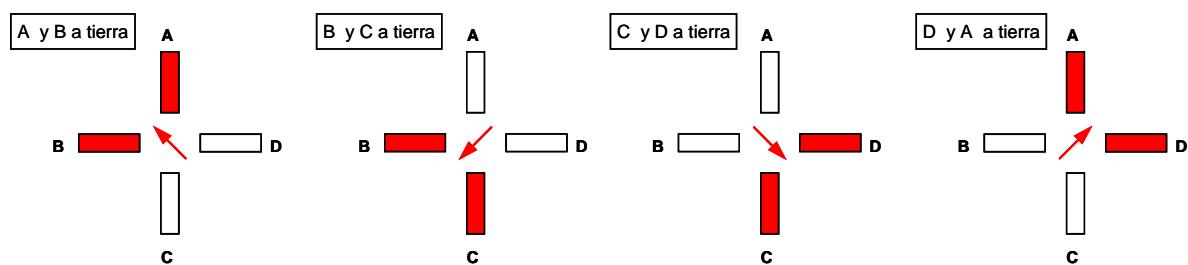


Figura 14.5: Secuencia de terminales que se ponen a tierra para obtener el giro del motor

Así que para girar el motor tenemos que poner a tierra de manera secuencial los terminales. De la figura 14.5 podemos obtener la secuencia de terminales que se ponen a tierra para obtener el giro del motor: AB-BC-CD-DA-AB-...

En la figura 14.6 se muestra la secuencia de los terminales que se tienen que poner a tierra. Se muestran los dos sentidos de giro.

terminal a tierra	tiempo								
	1	2	3	4	5	6	7	8	9
A	1			1	1		1	1	
B	1	1			1	1			1
C		1	1			1	1		
D			1	1			1	1	

terminal a tierra	tiempo								
	1	2	3	4	5	6	7	8	9
A	1	1				1	1		
B	1				1	1			1
C				1	1			1	1
D			1	1			1	1	

Figura 14.6: Secuencia de terminales que se ponen a tierra para obtener el giro del motor en ambos sentidos

Se pueden generar otras secuencias, poniendo a tierra un único terminal cada vez, o mezclando ambas alternativas (estas secuencias las puedes ver en [7]). Lo puedes hacer como ejercicio, pero para esta práctica utilizaremos las secuencias de la figura 14.6.

Antes de generar las secuencias, tenemos que mirar cuál es el tiempo mínimo que podemos estar en cada paso (o la frecuencia máxima). Esto varía según el motor y lo tienes que ver en las hojas de características. En nuestro motor, las hojas de características indican que a partir de 100 Hz empieza a bajar el par del motor<sup>24</sup>. De las pruebas que hemos hecho, a más de 200 Hz empieza a funcionar mal.

## 14.2. Generación de la secuencia de control con la FPGA

Ahora tenemos que generar las secuencias de la figura 14.6 con la FPGA. El bloque de control que haremos se muestra en la figura 14.7. Usaremos los pulsadores `BTN0` y `BTN3` para ordenar mover el motor en un sentido u otro. En principio no se deberían de pulsar a la vez, pero en caso de que se pulsen a la vez, daremos preferencia a un sentido de giro. Cuando no se pulsa ningún pulsador el motor deber estar parado.

Por ser un diseño síncrono que se hará con una máquina de estados, el circuito tendrá un reloj y un reset.

El circuito tendrá cuatro salidas (`A`, `B`, `C` y `D`) que cuando sean 1 darán la orden de poner el terminal correspondiente del motor a tierra. Cuando sean cero, se pondrán a la tensión de alimentación y no circulará corriente por el devanado del motor, y por tanto, no se activará.

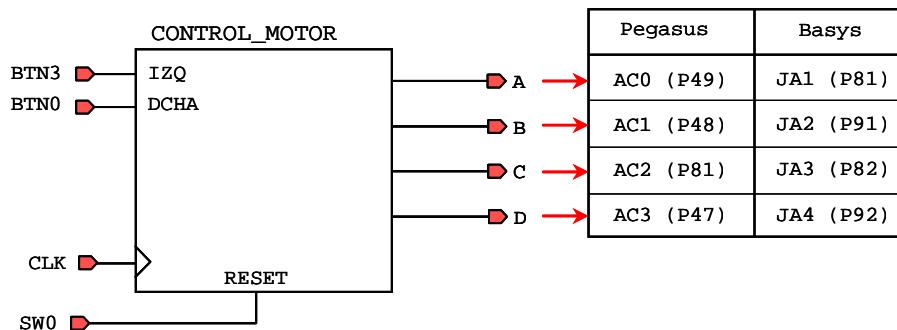


Figura 14.7: Bloque de control de motor de la FPGA

Las salidas las conectaremos a los conectores PMOD. La *Pegasus* tiene un único conector PMOD, que se ha llamado puerto accesorio (*accessory port*) en la figura 2.1. Los pines se muestran en la tabla de la derecha de la figura 14.7.

La *Basys* tiene cuatro conectores PMOD. Para esta práctica usaremos el conector de la izquierda (JA).

Ahora tenemos que diseñar el circuito y la máquina de estados que genere las secuencias a partir de las entradas que se han descrito. Intenta diseñarlo sin ayuda, y si ves que tienes dudas, consulta el diseño propuesto mostrado en la figura 14.8.

<sup>24</sup> También se ve influenciado por el tipo de electrónica de potencia. Con una electrónica específica se puede aumentar la frecuencia

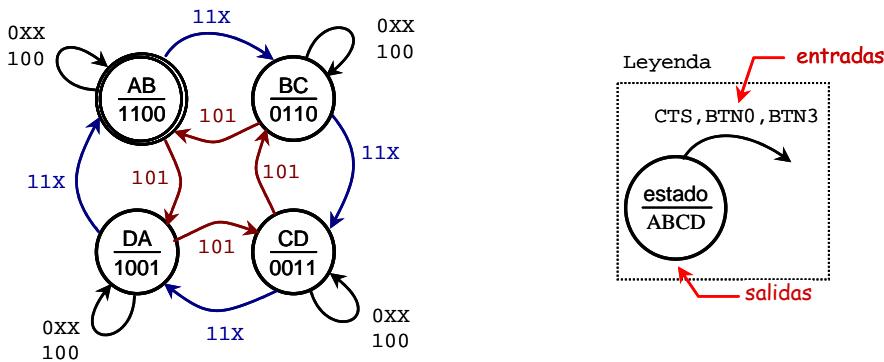


Figura 14.8: Esquema de la máquina de estados para el bloque de control de motor de la FPGA

Observa en la figura 14.8 que hay una señal CTS que no está en las entradas del bloque mostrado en la figura 14.7. Esta señal la tendremos que generar nosotros y será la que nos marque la frecuencia del motor. Para el motor de nuestro caso, será una señal de 100 Hz (una centésima de segundo). Esta señal la tenemos que generar de modo que esté activa durante un sólo ciclo de reloj en cada centésima de segundo.

Otra cosa que tienes que tener en cuenta es que para esta propuesta, las señales de entrada de los pulsadores BTN0 y BTN3 no se tienen que pasar por un detector de flanco, sino que se muestren continuamente, ya que quien hace de filtro es la señal CTS, pues está activa un sólo ciclo de reloj durante una centésima de segundo.

En la figura 14.8 las flechas que generan las dos secuencias de giro se han puesto de distinto color. Fíjate que la de color azul, la del BTN0 tiene preferencia sobre la de BTN3. En realidad es indiferente cuál escogas, pero tienes que escoger una secuencia sobre la otra.

Ahora tendrás que realizar el circuito en VHDL. Es recomendable que para que te sea más fácil depurar, las salidas A, B, C y D también las saques por cuatro LED. Incluso para depurar el circuito podrías bajar la frecuencia de giro y comprobar que efectivamente se generan las secuencias deseadas.

Una vez que tenemos el circuito y vemos que los LED se mueven como pensamos, nos queda hacer que las salidas A, B, C y D ordenen a los terminales del motor estar a tierra o a la tensión de alimentación según su valor. Esto lo veremos en el apartado siguiente.

### 14.3. Circuito de potencia para gobernar el motor

Podríamos pensar que si ponemos las cuatro salidas de la FPGA directamente conectadas a los terminales del motor, haríamos que los terminales se pongan a tierra cuando la FPGA lo mande. Desgraciadamente no es así, digamos que la FPGA no tiene *fuerza* suficiente como para obligar al motor a ponerse a cero, por tanto hay que pensar en hacerlo de otra manera. La FPGA no puede suministrar la potencia necesaria para mover un motor. Por otro lado, el motor va alimentado a 24 voltios (aunque puede funcionar a menos tensión) mientras que la FPGA se alimenta a 3,3 voltios, por tanto, las tensiones tampoco se corresponden.

En nuestra primera aproximación podríamos usar un transistor bipolar o MOSFET para cada terminal, de manera similar a como se mostraba en la figura 13.4. Haciéndolo de esta manera, el esquema nos quedaría como se muestra en la figura 14.9.

Analizando el esquema podemos ver que cuando una salida de la FPGA (por ejemplo, `FPGA_A`) tiene un valor lógico de 1 (3,3 V), circulará corriente por la base del transistor y

entrará en saturación. Esto provocará que el terminal del motor TA esté a una tensión cercana a cero (según la caída de tensión entre colector y emisor del transistor).

Por otro lado, si hay un cero lógico a la salida de la FPGA, habrá cero voltios en la base del transistor y no circulará corriente por la base. Por lo tanto el transistor estará en corte y el terminal del motor TA estará a  $V_{cc}$ .

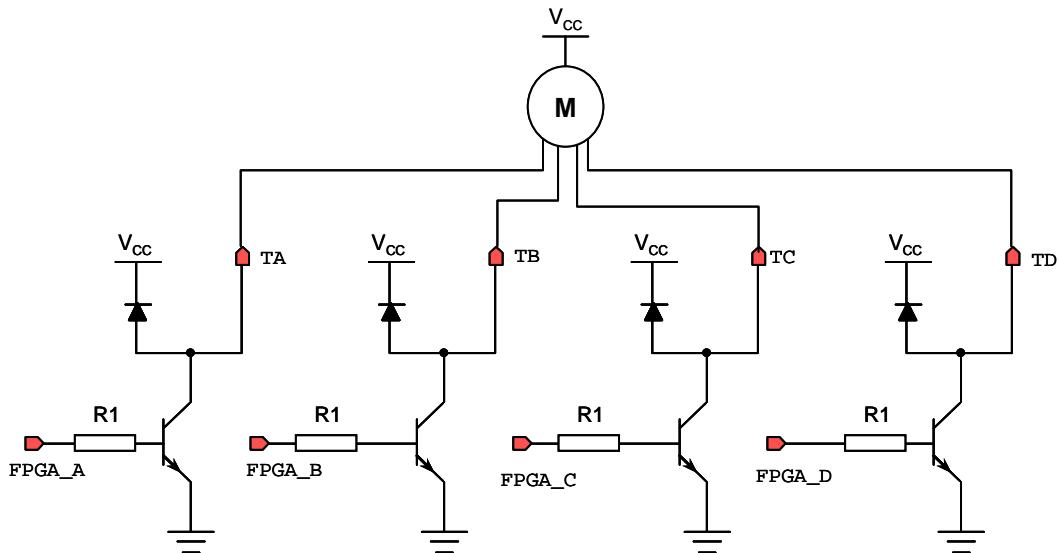


Figura 14.9: Posible esquema de la electrónica de potencia para manejar el motor

Este circuito podría funcionar bien, sin embargo hay que tener cuidado porque la FPGA va alimentada con una fuente de tensión distinta que la del motor (3,3V la FPGA y 24V el motor). Para evitar que corrientes o tensiones altas lleguen a la FPGA desde el motor, es recomendable aislar la parte de la FPGA de la del motor. Para aislar ambas partes podemos usar optoacopladores. Los optoacopladores utilizan fotodiodos y fototransistores para transmitir información proporcionando un aislamiento eléctrico entre los circuitos de entrada y salida. La figura 14.10 muestra un circuito que se ha probado que funciona.

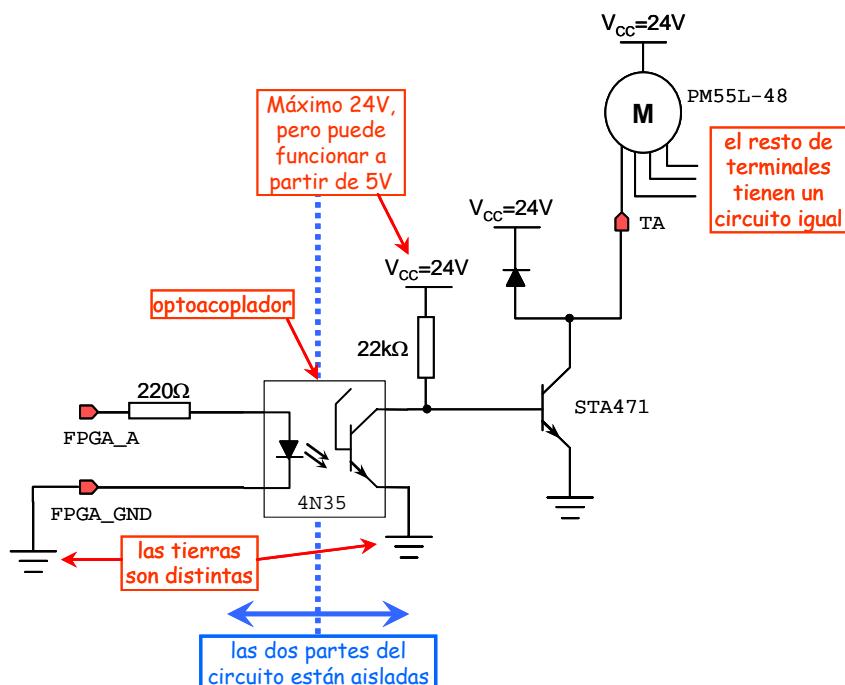


Figura 14.10: Esquema del circuito con optoacopladores

Los diodos que hay entre el colector del transistor y la alimentación  $v_{2cc}$  son para que la corriente que circula por los devanados del motor tenga un camino de circulación cuando el transistor entra en corte, evitando con esto sobretensiones.

El esquema muestra los valores concretos en las resistencias. El transistor que se ha utilizado es el STA471 de *Sanken* [20]. Este transistor es un *darlington* y se ha usado porque, al igual que el motor, se ha reutilizado de la impresora. Este integrado tiene los cuatro transistores. Con pocas modificaciones se podría haber utilizado otro similar como un TIP120 (*darlington*) o un IRL540 (MOSFET).

El optoacoplador utilizado es el 4N35, pero se pueden utilizar otros similares como el ACPL-847 que tiene como ventaja que tiene cuatro optoacopladores integrados en el circuito integrado. Fíjate que tal como está hecho el circuito con los optoacopladores, la señal de la FPGA se invierte, así que habría que negar las salidas de la FPGA: invertir las salidas de la máquina de estados, o si no, saber que el motor va a girar en sentido contrario a lo que haría sin optoacopladores.

Como el transistor STA471 no nos ha sido fácil de encontrar para comprar (lo cogimos de la propia impresora), hemos realizado una variante del circuito con un integrado SN754410 que incluye dos medios puentes en H. La variante del circuito se muestra en la figura 14.11. El *driver* del motor del SN754410 tiene una habilitación que se pone a uno para dejarlo habilitado.

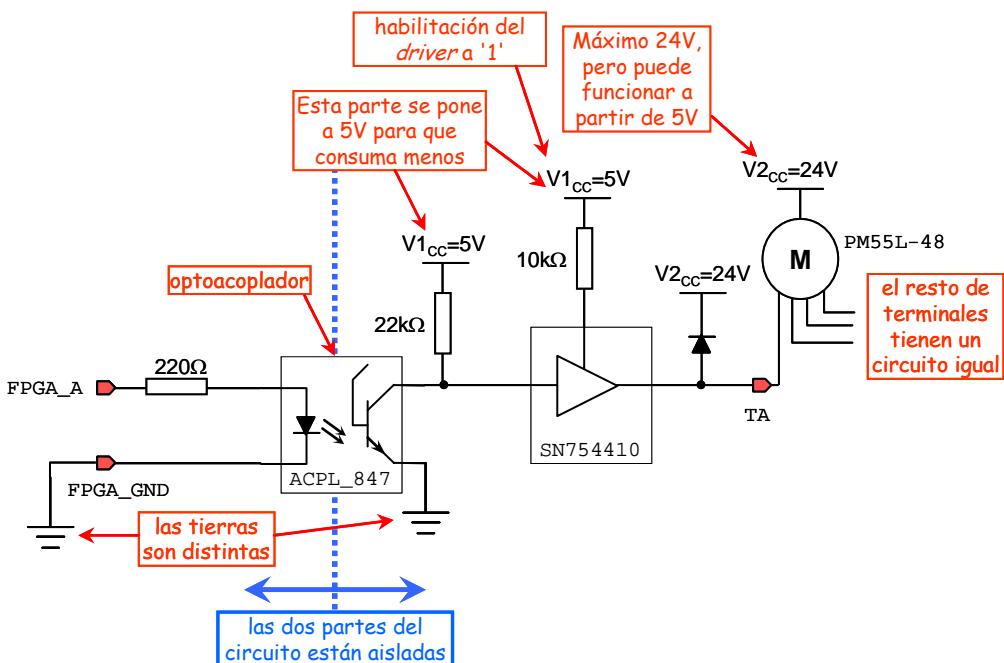


Figura 14.11: Esquema del circuito usando el circuito integrado SN754410

#### 14.4. Conclusiones

En este diseño nos hemos enfrentado no sólo al diseño de la parte digital, sino a las dificultades que aparecen al interactuar con dispositivos externos. Hemos realizado el diseño de la electrónica de potencia sin adentrarnos en conceptos teóricos y sin analizar alternativas, pues daría para un libro entero. La alternativa a realizar el diseño es comprar módulos con la electrónica de potencia y que se pueden conectar a la FPGA.



## 15. Piano electrónico<sup>25</sup>

En esta práctica realizaremos un piano electrónico. Para ello generaremos un conjunto de señales digitales periódicas de frecuencias audibles y las amplificaremos para escucharlas por un altavoz. La parte analógica de amplificación se puede adquirir<sup>26</sup> o la podemos diseñar nosotros mismos si tenemos conocimientos básicos de electrónica analógica. La parte digital, con lo que ya sabemos, no debería ser ningún problema.

En la versión más básica (figura 15.1), la selección de las notas se hará a través de los pulsadores o interruptores de la FPGA. Según que pulsador se haya seleccionado, la salida de la FPGA que va al amplificador de audio tendrá una frecuencia determinada.

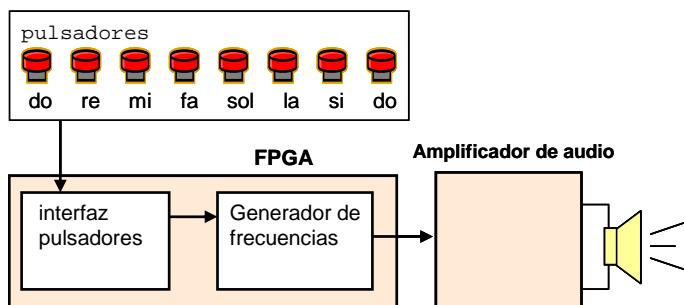


Figura 15.1: Esquema del circuito de piano electrónico básico

La frecuencia de las señales que queremos generar dependerá de la octava que escojamos, lo que hará que sean más o menos graves los sonidos que generemos. La tabla 15-1 muestra distintas frecuencias de las notas.

La frecuencia de las señales que queremos generar dependerá de la octava que escojamos, lo que hará que sean más o menos graves los sonidos que generemos. La tabla 15-1 muestra distintas frecuencias de las notas.

Nota Hz	LA	SI	DO	RE	MI	FA	SOL	LA
	220,00	246,94	261,63	293,67	329,63	349,23	392,00	440,00
	440,00	493,88	523,25	587,33	659,26	698,46	783,99	880,00
	880,00	987,77	1046,5	1174,7	1318,5	1396,9	1568,0	1760,0

Tabla 15-1:Frecuencias de las notas en distintas octavas

Realizar un circuito que genere una señal periódica lo hemos hecho en la práctica de contadores (práctica 6), así que no deberíamos tener mucho problema. La señal generada no debería estar la mitad de tiempo a uno y la otra mitad a cero, esto es, no debería estar a uno un único ciclo de reloj.

Finalmente la señal salida de la FPGA servirá de entrada a la etapa de amplificación, y a su vez, la salida del amplificador se conectaría a una resistencia de carga, que en nuestro

<sup>25</sup> Esta práctica ha sido desarrollada en el departamento de Tecnología Electrónica de manera conjunta por los profesores: Belén Arredondo, Susana Borromeo, Joaquín Vaquero, Norberto Malpica y Felipe Machado. La implantación de esta práctica en clases se ha detallado en el artículo [10]

<sup>26</sup> PmodAMP1: <http://www.digilentinc.com/Products/Detail.cfm?&Prod=PMOD-AMP1>

caso un altavoz. La figura muestra el esquema más general del amplificador. La primera etapa del amplificador eleva los niveles de tensión y la segunda etapa amplifica la corriente que será entregada al altavoz.

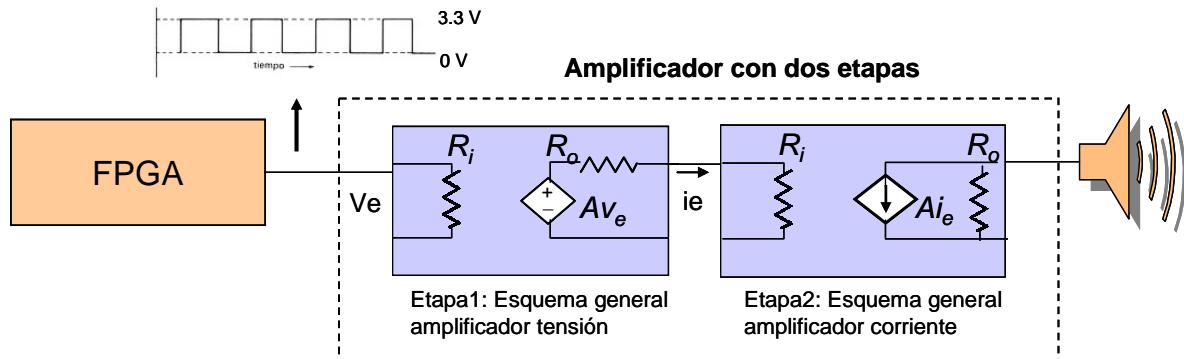


Figura 15.2: Esquema general del circuito de amplificación

Es importante señalar que el diseño más básico del amplificador puede no necesitar etapa amplificadora de tensión. Sin embargo, puede ser interesante diseñar una primera etapa con ganancia en tensión variable (utilizando un simple potenciómetro) y así conseguir un nivel de voltaje variable a la salida.

Para realizar el diseño del amplificador debemos tener en cuenta las especificaciones del problema de diseño:

Para las características de entrada del amplificador hay que tener en cuenta que la señal de entrada al amplificador (la salida de la FPGA) es una señal cuadrada cuya amplitud varía entre 0 y 3,3 V.

Para diseñar las características de salida del amplificador hay que tener en cuenta que la resistencia de carga (altavoz) es  $8 \Omega$  y que los altavoces utilizados pedirán una potencia entre 0,5 - 0,25 W.

Finalmente, tenemos que diseñar un ancho de banda adecuado para que el amplificador funcione correctamente en todas las frecuencias de las notas. En el caso más general, podemos diseñar un amplificador que opere correctamente en el rango de frecuencias de las tres octavas, es decir, que las frecuencias de corte inferior (y superior) se sitúen una década por debajo (y por encima) de la nota con frecuencia menor, 220 Hz (y la nota con frecuencia mayor, 1760 Hz). En el rango de frecuencias en que trabajamos, la frecuencia de corte superior no supone ningún problema ya que ésta viene determinada por las capacidades internas del transistor, cuyos valores son muy pequeños, del orden pF, resultando frecuencias de corte superior mucho más altas de las requeridas. Para diseñar la frecuencia de corte inferior necesitamos elegir cuidadosamente los condensadores de acoplado de la entrada y la salida del amplificador, ya que éstos determinarán nuestra frecuencia de corte a bajas frecuencias.

## 15.1. Ampliaciones

Hay muchas ampliaciones posibles del circuito, tener una melodía predeterminada, mostrar por los *displays* de siete segmentos la nota que se está tocando, poder seleccionar distintas octavas, y muchas otras opciones que se te pueden ocurrir mientras lo diseñas.

## Problemas teóricos

En esta sección se incluye la resolución de problemas teóricos de diseño digital. Te recomendamos que los intentes hacer sin ver la solución, y que mires la solución una vez lo hayas intentado para comprobar el resultado. Los problemas de electrónica digital parecen muy fáciles si tienes la solución al lado, pero lo difícil es pensarlos.

Estos problemas han sido problemas de exámenes de la asignatura Electrónica Digital II de la carrera de Ingeniería de Telecomunicación de la Universidad Rey Juan Carlos. Por ser problemas de exámenes, el enunciado está simplificado, para adecuarlos a la resolución en un tiempo limitado, por tanto, hay muchas simplificaciones y en muchos casos se asumen comportamientos ideales.



## 16. Reproductor MP3

Este problema se puso en el examen del 30 de enero de 2009

---

### 16.1. Enunciado

Diseñar un circuito para controlar un reproductor de MP3. Para el control, el dispositivo tiene un único pulsador (P) y tiene dos salidas que gobiernan la reproducción normal (PLAY) y el avance rápido (FW). Estas salidas funcionan a nivel alto y cuando están las dos a cero no se reproduce ni se avanza (como si estuviese en pausa).

La reproducción normal (PLAY) se consigue cuando se presiona el pulsador durante menos de un segundo. Al soltar el pulsador el circuito se mantiene en reproducción normal. Si se está en reproducción normal (PLAY) y se pulsa P por menos de un segundo se detiene la reproducción.

El avance rápido (FW) se consigue cuando se presiona el pulsador durante más de un segundo, y se debe mantener presionado para que continúe el avance. Al soltarlo, se vuelve a la reproducción normal (PLAY).

Concretando, el circuito debe funcionar de la siguiente manera:

1. Al comenzar, el circuito se encuentra en estado inicial en el que no se reproduce ni se avanza.  $\text{PLAY}='0'$  y  $\text{FW}='0'$ .
2. Estando en el estado inicial, si se pulsa P pueden suceder dos cosas dependiendo del tiempo que se mantenga pulsado:
  - 2.1. Si se ha mantenido pulsado P menos de un segundo, al soltar se debe pasar a la reproducción normal ( $\text{PLAY}='1'$ ).
  - 2.2. Si se mantiene pulsado P más de un segundo, se debe pasar al avance rápido (FW) y se mantendrá este avance rápido hasta soltar el pulsador. Cuando se suelte se pasa a la reproducción normal (PLAY).
3. Cuando el circuito está en reproducción normal (PLAY) se debe mantener este modo de reproducción mientras no se pulse P. Si se pulsa P, pueden pasar dos cosas:
  - 3.1. Si se ha mantenido pulsado P menos de un segundo, al soltar se debe ir al estado inicial. (durante el tiempo que se mantiene pulsado se sigue reproduciendo la música:  $\text{PLAY}='1'$ )
  - 3.2. Si se mantiene pulsado P más de un segundo, se debe pasar al avance rápido (FW) y se mantendrá este avance hasta soltar el pulsador. Cuando se suelte el pulsador se vuelve a pasar a la reproducción normal (PLAY). *Nota:* antes de que pase el segundo, se sigue reproduciendo la música:  $\text{PLAY}='1'$

Datos adicionales del sistema:

- El pulsador no tiene rebotes
- El reloj del circuito (Clk) va a 10 MHz
- Existe un pulsador para resetear asíncronamente (Reset), pero no se debe de presionar durante el funcionamiento normal del circuito

**Se pide:**

- Dibujar los bloques internos del circuito, con entradas, salidas y señales intermedias
- Dibujar el diagrama de transición de estados de la máquina de estados que controla el circuito. Realizarla como máquina de Moore.
- La tabla de estados siguientes y salidas
- La tabla de excitación de los biestables usando biestables J-K
- Ecuación simplificada **sólo** para la salida PLAY
- Realizar el modelo VHDL de los procesos de la máquina de estados

## 16.2. Solución

### 16.2.1. Bloques internos del circuito

Para realizar el circuito necesitamos una máquina de estados y un temporizador de un segundo con habilitación (enable). El esquema de este circuito se muestra en la figura 16.1.

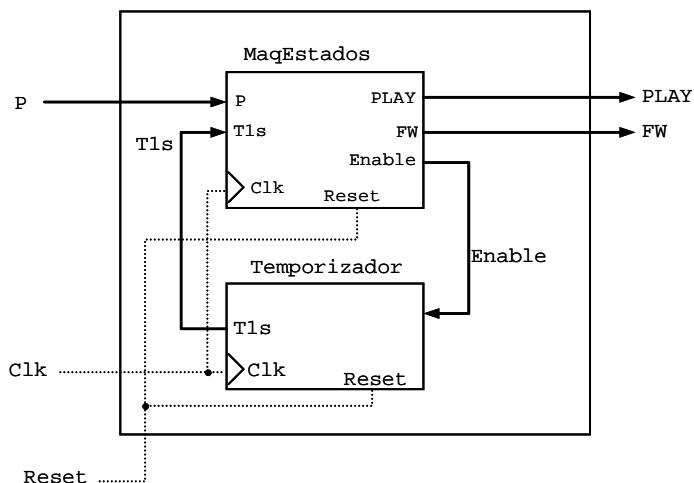


Figura 16.1: Bloques internos del circuito

El temporizador lo necesitamos para saber si se presiona el pulsador  $P$  durante más de un segundo. Por lo tanto, la máquina de estados tendrá una entrada más que indica el fin de la cuenta ( $T1s$ ). También necesitamos una salida más que va a habilitar el temporizador (Enable). Cuando el temporizador esté deshabilitado no podrá estar activa la señal de fin de cuenta  $T1s$ .

### 16.2.2. Diagrama de transición de estados

Nos piden realizar el diagrama de la máquina de estados por Moore.

La máquina de estados tiene dos entradas:

- El pulsador  $P$
- La señal que indica que ha pasado un segundo  $T1s$  (fin de la cuenta)

Y tres salidas

- Orden de play (PLAY)
- Orden de avance rápido (FW).
- Habilitación del temporizador (Enable)

El orden de las entradas y salidas en el diagrama será el siguiente (figura 16.2).



Figura 16.2: Orden de entradas y salidas en la representación de la máquina de estados

Fíjate que es una máquina de Moore, por lo que las salidas están en el propio estado.

Inicialmente partimos del estado `INICIAL`, en donde `PLAY= '0'` y `FW= '0'`, es como estar en *pause*. En este estado el contador está deshabilitado, y nos mantendremos aquí hasta que se pulse `P` (figura 16.3).

Como el contador está deshabilitado, nunca llegará la señal de que ha pasado un segundo (`T1s`), por lo que siempre será cero.

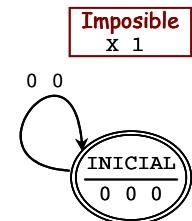


Figura 16.3: Estado inicial

¿Qué sucede si se pulsa `P` estando en el estado `INICIAL`? depende del tiempo en que esté pulsado, así que tendremos que ir a un estado en el que mientras esté pulsado `P` habilitemos la cuenta y veamos si se mantiene pulsado más de un segundo o no (figura 16.4).

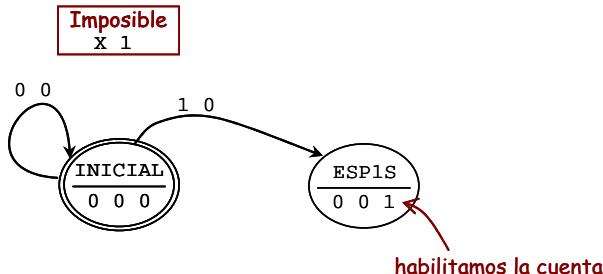


Figura 16.4: Transición a un estado de espera

En este estado de espera (`ESP1S`), esperaremos a que (figura 16.5):

- O bien llegue el final de cuenta sin haber soltado el pulsador (ha pasado más de un segundo) y entonces hay que ir al avance rápido (`FW`).
- O bien se suelte el pulsador antes del fin de cuenta, y entonces hay que ir al `PLAY`.
- Podemos decir que es prácticamente imposible que se suelte el pulsador en el mismo instante en el que se termina la cuenta. Sin embargo, en tal caso optaremos por darle prioridad al play (no está especificado)

Si no ocurre nada de lo anterior nos mantendremos en dicho estado. Esto es, si no se ha soltado el pulsador (`P='1'`) y no se ha llegado al fin de cuenta (`T1s='0'`).

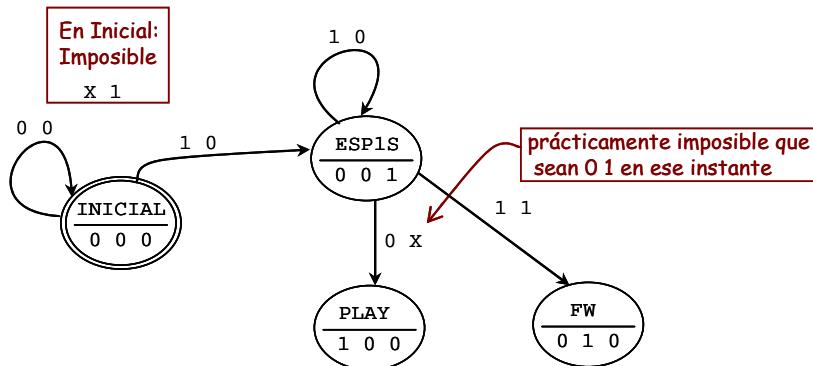


Figura 16.5: Transición a estado de reproducción o avance

Estando en **PLAY** no puede llegar el fin de cuenta porque está deshabilitado el contador. Y nos mantendremos en este estado mientras no se vuelva a pulsar  $P$ .

Estando en **FW** no puede llegar el fin de cuenta porque está deshabilitado el contador. Y nos mantendremos en este estado mientras no se suelte  $P$ . Estas dos posibilidades se han añadido en la figura 16.6.

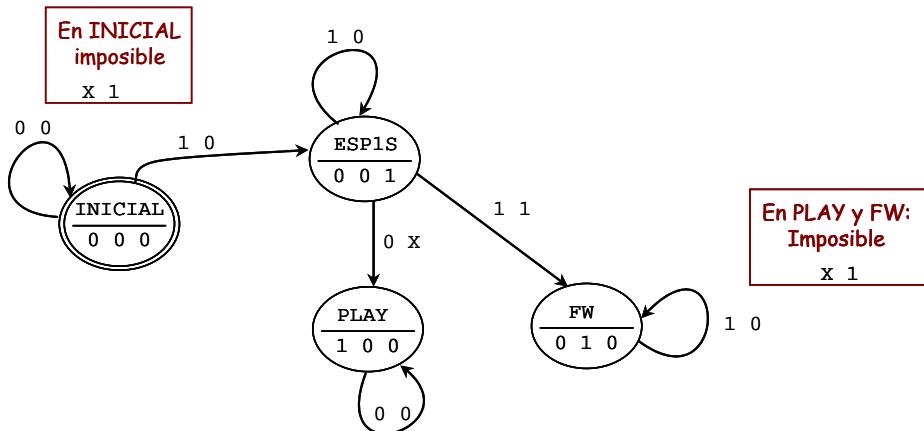


Figura 16.6: Estado de reproducción y avance

Estando en **FW** cuando se suelte el pulsador  $P$ , el sistema va al **PLAY** (figura 16.7)

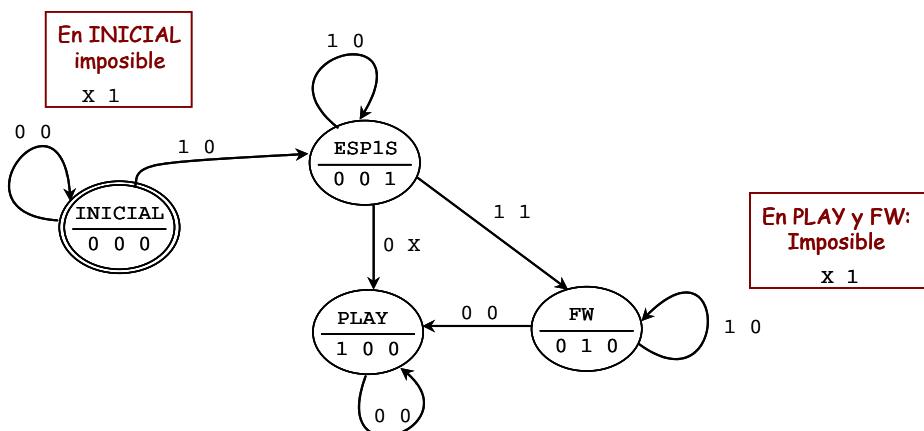


Figura 16.7: Transición desde el estado de avance al estado de reproducción

Estando en **PLAY** cuando se pulsa  $P$ , tenemos que comprobar de nuevo cuánto tiempo se pulsa. Si se pulsa más de un segundo vamos a **FW** y si es menos, vamos a **INICIAL** (*pause*). Así pues, necesitamos otro estado de espera en el que volveremos a habilitar el contador.

A este estado le llamamos `ESP1S2` (*espera un segundo, 2*). Podemos decir también que es prácticamente imposible que en el mismo instante que soltemos el pulsador (`P='0'`) haya justo terminado la cuenta (`T1S='1'`). En este hipotético caso decidimos ir al estado `INICIAL` (haber ido al estado `FW` hubiese sido igualmente válido).

En la figura 16.8 se muestra el diagrama final de la máquina de estados.

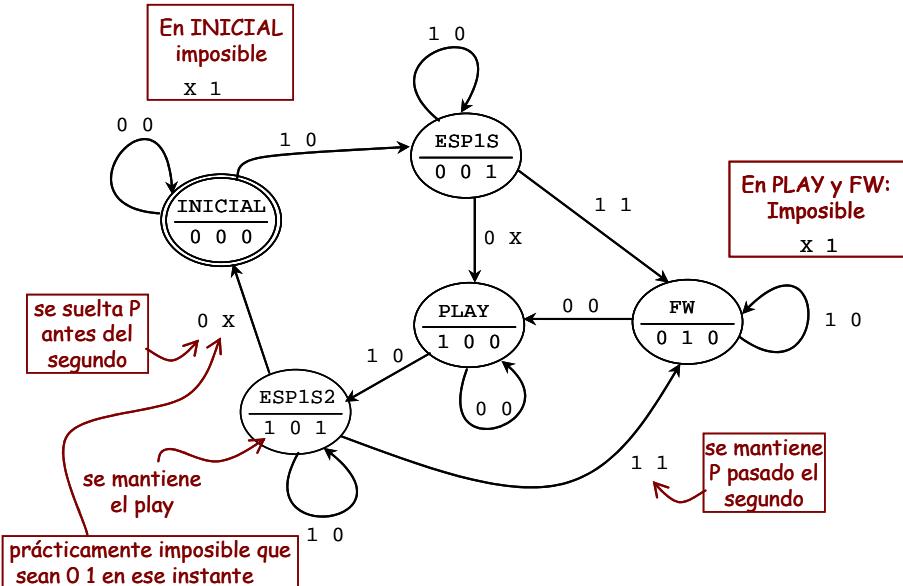


Figura 16.8: Diagrama final de la máquina de estados

### 16.2.3. Tabla de estados siguientes y salidas

La tabla de estados siguientes y salidas la obtenemos fijándonos en el diagrama de transición de estados que acabamos de hacer (figura 16.8).

Antes de nada, tenemos que asignar una codificación a cada estado. Como tenemos 5 estados necesitamos 3 biestables (ya que con dos biestables sólo podemos codificar cuatro estados). Escogemos una codificación cualquiera, por ejemplo la resultante de tomar números consecutivos. Seguramente habrá codificaciones más óptimas que reduzcan las transiciones de un estado a otro (pero no nos importa). La codificación escogida se muestra en la tabla 16.1.

Estado	Q2	Q1	Q0
Inicial	0	0	0
Esp1s	0	0	1
FW	0	1	0
PLAY	0	1	1
Esp2s	1	0	0

Tabla 16.1:  
Codificación de estados

Ahora rellenamos la tabla de los estados siguientes y salidas (tabla 16.2):

**Moore: sólo dependen del estado**

Estado actual			Entradas			Salidas			Estado siguiente		
			P	T1s	P1	FW	En	$Q_2^{t+1}$	$Q_1^{t+1}$	$Q_0^{t+1}$	
Inicial	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	1	0	0	0	X	X	X
	0	0	0	1	0	0	0	0	0	0	1
	0	0	0	1	1	0	0	0	X	X	X
Esp1s	0	0	1	0	0	0	0	1	0	1	1
	0	0	1	0	1	0	0	1	0	1	1
	0	0	1	1	0	0	0	1	0	0	1
	0	0	1	1	1	0	0	1	0	1	0
Fw	0	1	0	0	0	0	1	0	0	1	1
	0	1	0	0	1	0	1	0	X	X	X
	0	1	0	1	0	0	1	0	0	1	0
	0	1	0	1	1	0	1	0	X	X	X
Play	0	1	1	0	0	1	0	0	0	1	1
	0	1	1	0	1	1	0	0	X	X	X
	0	1	1	1	0	1	0	0	1	0	0
	0	1	1	1	1	1	0	0	X	X	X
Esp1s2	1	0	0	0	0	1	0	1	0	0	0
	1	0	0	0	1	1	0	1	0	0	0
	1	0	0	1	0	1	0	1	1	0	0
	1	0	0	1	1	1	0	1	0	1	0
Estados imposibles	1	X	1	X	X	X	X	X	X	X	X
	1	1	0	X	X	X	X	X	X	X	X

Tabla 16.2: Tabla de estados siguientes y salidas

#### 16.2.4. Tabla de excitación de los biestables para biestables J-K

La tabla de excitación de los biestables la podríamos haber rellenado en la tabla anterior. Para saber las ecuaciones de las entradas de los biestables necesitamos saber qué necesitamos para lograr cada una de las transiciones. Como nos piden hacerlo con biestables J-K, ponemos su tabla, y a partir de ella obtenemos las entradas J y K para las cuatro posibles transiciones (tabla 16.3)

J	K	$Q(t+1)$	Q(t)	J	K
0	0	$Q(t)$	0	0	X
0	1	0	0	1	X
1	0	1	1	X	1
1	1	$\overline{Q(t)}$	1	X	0

Tabla 16.3: Tabla de entradas necesarias para obtener una transición en biestables J-K

Ahora, nos fijamos en la tabla de la tabla 16.2 para ver en cada caso qué transiciones hay y en consecuencia, qué entradas necesitamos. En la tabla 16.4 tenemos lo que nos pide este apartado.

	Estado actual			Entradas			Salidas			Estado siguiente			Entradas en los biestables para conseguir las transiciones $Q_i \rightarrow Q_i^{t+1}$					
	$Q_2$	$Q_1$	$Q_0$	$P$	$T1s$	$P1$	$Fw$	$En$	$Q_2^{t+1}$	$Q_1^{t+1}$	$Q_0^{t+1}$	$J_2$	$K_2$	$J_1$	$K_1$	$J_0$	$K_0$	
Inicial	0	0	0	0	0	0	0	0	0	0	0	0	X	0	X	0	X	
	0	0	0	0	1	0	0	0	X	X	X	X	X	X	X	X	X	
	0	0	0	1	0	0	0	0	0	0	1	0	X	0	X	1	X	
	0	0	0	1	1	0	0	0	X	X	X	X	X	X	X	X	X	
Esp1s	0	0	1	0	0	0	0	1	0	1	1	0	X	1	X	X	0	
	0	0	1	0	1	0	0	1	0	1	1	0	X	1	X	X	0	
	0	0	1	1	0	0	0	1	0	0	1	0	X	0	X	X	0	
	0	0	1	1	1	0	0	1	0	1	0	0	X	1	X	X	1	
Fw	0	1	0	0	0	0	1	0	0	1	1	0	X	X	0	1	X	
	0	1	0	0	1	0	1	0	X	X	X	X	X	X	X	X	X	
	0	1	0	1	0	0	1	0	0	1	0	0	X	X	0	0	X	
	0	1	0	1	1	0	1	0	X	X	X	X	X	X	X	X	X	
Play	0	1	1	0	0	1	0	0	0	1	1	0	X	X	0	1	X	
	0	1	1	0	1	1	0	0	X	X	X	X	X	X	X	X	X	
	0	1	1	1	0	1	0	0	1	0	0	1	X	X	1	X	1	
	0	1	1	1	1	1	0	0	X	X	X	X	X	X	X	X	X	
Esp1s2	1	0	0	0	0	1	0	1	0	0	0	0	X	1	0	X	0	
	1	0	0	0	1	1	0	1	0	0	0	0	X	1	0	X	0	
	1	0	0	1	0	1	0	1	1	0	0	0	X	0	0	X	0	
	1	0	0	1	1	1	0	1	0	1	0	0	X	1	1	X	0	
Estados imposibles	1	X	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
	1	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	

Tabla 16.4: Tabla de excitación de los necesarias para obtener una transición en biestables J-K

### 16.2.5. Ecuación simplificada para la salida Play

De toda la tabla 16.4 nos piden la ecuación simplificada para la salida PLAY. Para ello realizamos Karnaugh. Por ser de Moore, las entradas no hacen falta (se simplificarán). En la figura 16.9 se muestra el diagrama de Karnaugh y la ecuación de la salida PLAY.

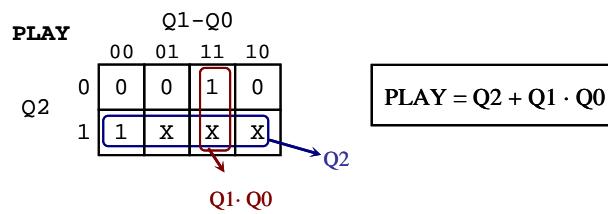


Figura 16.9: Diagrama de Karnaugh y ecuación de la salida Play

### 16.2.6. Modelo VHDL

Aunque sólo se piden los tres procesos de la máquina de estados, se incluye todo el diseño VHDL: la entidad (código 16-1) y la arquitectura (código 16-2).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity mp3 is
    Port ( Reset : in STD_LOGIC;
           Clk : in STD_LOGIC;
           P : in STD_LOGIC;
           Play : out STD_LOGIC;
           Fw : out STD_LOGIC);
end mp3;
```

Código 16-1: Entidad del circuito MP3

```

architecture Behavioral of mp3 is
-- el estado no se puede llamar PLAY ni FW para diferenciarlos de los puertos
type estados is (INICIAL, ESP1S, ePLAY, eFW, ESP1S2);
signal estado_actual, estado_siguiente : estados;
signal T1S, enable : std_logic;
signal cuenta : unsigned (23 downto 0); -- 24 bits mayor que 10 millones
constant cfincuenta : natural := 10**7; -- contar 10 millones
begin

----- PROCESO COMBINACIONAL DEL ESTADO -----
P_COMB_ESTADO: Process (estado_actual, T1S, P)
begin
    case estado_actual is
        when INICIAL =>
            if P = '1' then
                estado_siguiente <= ESP1S;
            else
                estado_siguiente <= estado_actual;
            end if;
        when ESP1S =>
            if P = '0' then
                estado_siguiente <= ePLAY;
            elsif T1S = '1' then
                estado_siguiente <= eFW;
            else
                estado_siguiente <= estado_actual;
            end if;
        when ePLAY =>
            if P = '1' then
                estado_siguiente <= ESP1S2;
            else
                estado_siguiente <= estado_actual;
            end if;
        when eFW =>
            if P = '0' then
                estado_siguiente <= ePLAY;
            else
                estado_siguiente <= estado_actual;
            end if;
        when ESP1S2 =>
            if P = '0' then
                estado_siguiente <= INICIAL;
            elsif T1S = '1' then
                estado_siguiente <= eFW;
            else
                estado_siguiente <= estado_actual;
            end if;
    end case;
end process;

----- PROCESO COMBINACIONAL DE LAS SALIDAS (MOORE) -----
P_COM_SALIDAS: Process (estado_actual)
begin
    case estado_actual is
        when INICIAL =>
            Play <= '0';
            Fw <= '0';
            enable <= '0';
        when ESP1S =>
            Play <= '0';
            Fw <= '0';
            enable <= '1';
        when ePLAY =>
            Play <= '1';
            Fw <= '0';
            enable <= '0';
        when eFW =>
            Play <= '0';
            Fw <= '1';
            enable <= '0';
        when ESP1S2 =>
            Play <= '1';
            Fw <= '0';
            enable <= '1';
    end case;
end process;

```

```

----- PROCESO SECUENCIAL -----
P_SEQ: Process (Reset, Clk)
begin
  if Reset = '1' then
    estado_actual <= INICIAL;
  elsif Clk'event and Clk='1' then
    estado_actual <= estado_siguiente;
  end if;
end process;

----- TEMPORIZADOR -----
P_Tempo: Process (Reset, Clk)
begin
  if Reset = '1' then
    cuenta <= (others => '0');
    T1S <= '0';
  elsif Clk'event and Clk='1' then
    if enable = '1' then
      if cuenta < cfincuenta then
        cuenta <= cuenta + 1;
        T1S <= '0';
      else
        cuenta <= (others => '0');
        T1S <= '1';
      end if;
    else
      cuenta <= (others => '0');
      T1S <= '0';
    end if;
  end if;
end process;
end Behavioral;

```

Código 16-2: Arquitectura del circuito MP3

### 16.3. Solución alternativa

Suele haber varias alternativas para diseñar un circuito digital. En el examen, varios alumnos plantearon una solución similar a la siguiente (figura 16.10):

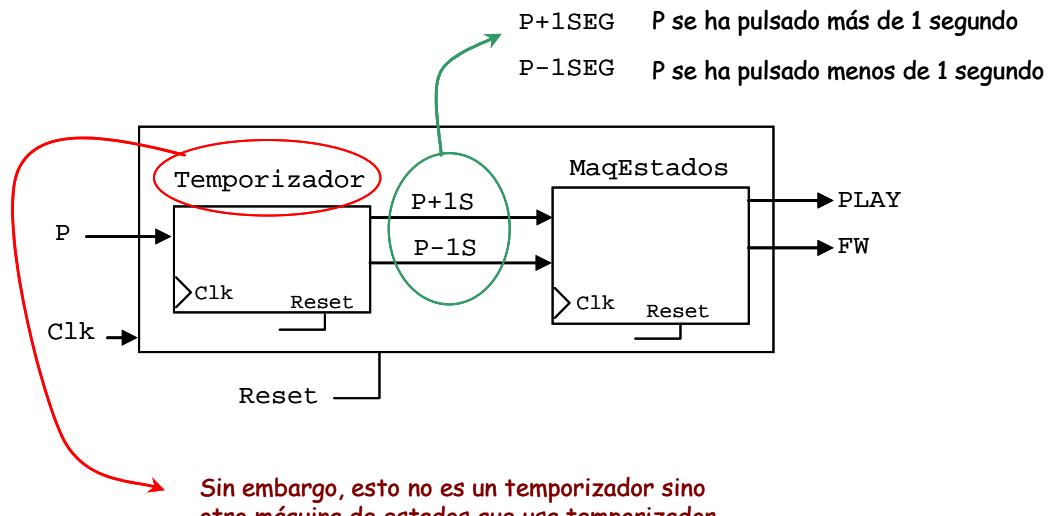


Figura 16.10: Diagrama de bloques de la solución alternativa

Sin embargo, fíjate que el primer bloque (el llamado *temporizador*) es mucho más que un temporizador. Es una máquina de estados. Por lo tanto, al hacerlo de esta manera se deben definir las dos máquinas de estados:

El esquema tendría que ser el mostrado en la figura 16.11 (o similar).

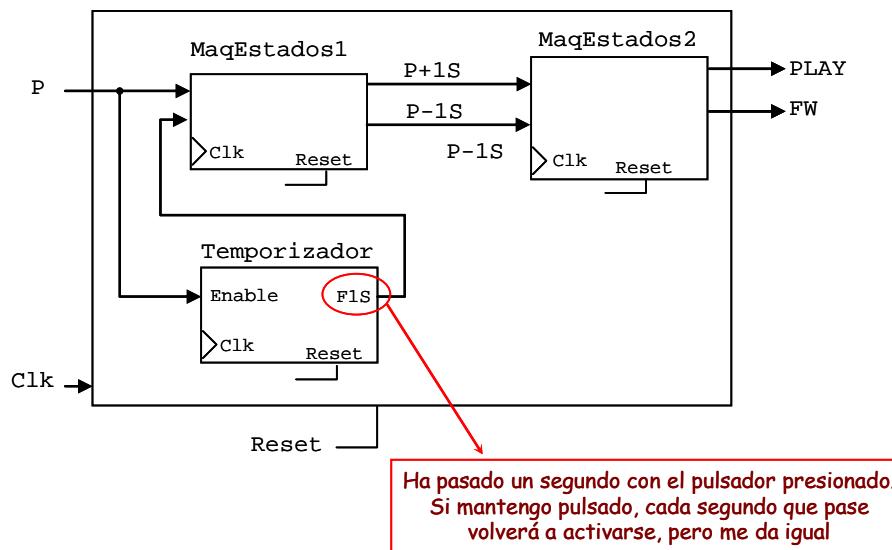


Figura 16.11: Diagrama de bloques más detallado de la solución alternativa

Por tanto, habría que definir las dos máquinas de estados.

La primera máquina de estados se muestra en la figura 16.13. La interpretación de las entradas y salidas la tenemos en la figura 16.12 .

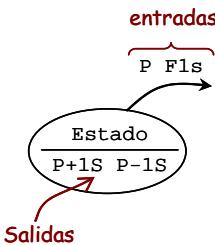


Figura 16.12: Orden de entradas y salidas en la representación de la primera máquina de estados

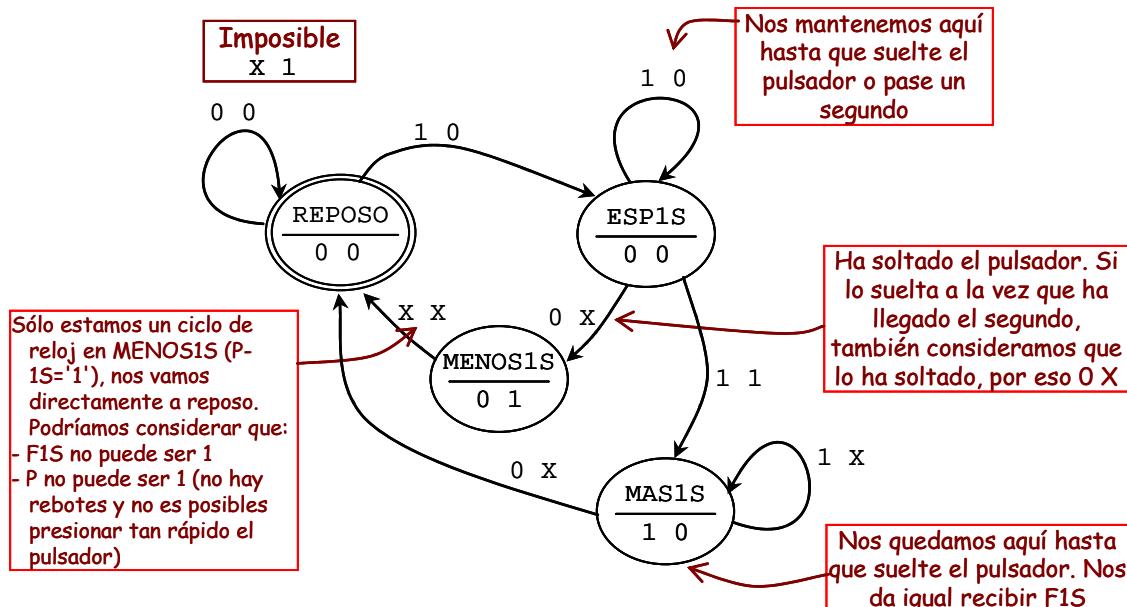


Figura 16.13: Diagrama de la primera máquina de estados de la solución alternativa

Es importante notar que la señal  $P-1S$  es activa durante un sólo ciclo de reloj, mientras que la señal  $P+1S$  está activa todo el tiempo que esté pulsado  $P$  a partir de que ha pasado un segundo.

La segunda máquina de estados se muestra en la figura 16.15. En la figura 16.14 tenemos la interpretación de las entradas y salidas. Esta máquina de estados fue propuesta por muchos alumnos, pero es necesario haber incluido la primera, pues como puede apreciarse, la primera es algo más complicada y no la segunda sola no funcionaría como indica el enunciado.



Figura 16.14: Orden de entradas y salidas en la representación de la segunda máquina de estados

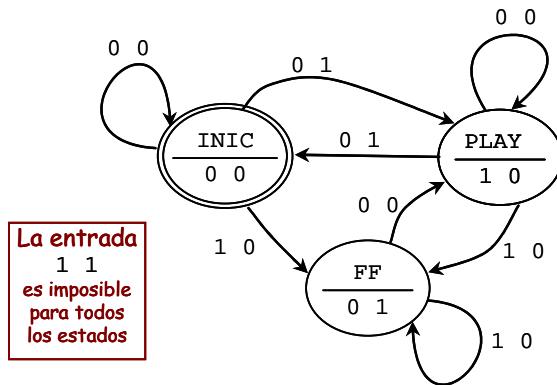


Figura 16.15: Diagrama de la segunda máquina de estados de la solución alternativa



## 17. Visualización del reproductor MP3

Este problema se puso en el examen del 30 de enero de 2009 y hace referencia al problema del capítulo 16.

### 17.1. Enunciado

Para el ejercicio del capítulo 16, ahora queremos mostrar por 4 *displays* de siete segmentos el estado en que estamos. Para simplificar, para este ejercicio **suponemos que sólo tenemos tres estados**: PAUSA, PLAY, FW (avance rápido). Y queremos mostrar los siguientes caracteres por los 4 *displays* según el estado en que estemos.

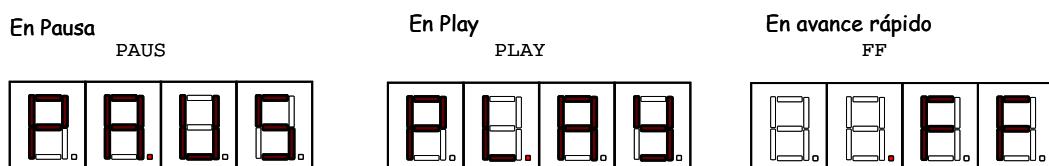


Figura 17.1: Texto que deben de mostrar los displays de siete segmentos

El funcionamiento de los *displays* es igual que el de la placa de prácticas. Esto es, tenemos 4 ánodos que controlan el encendido de cada uno de los *displays*: AN3, AN2, AN1, AN0. Recuerda que funcionan a nivel bajo. Y tenemos 7 puertos para los siete segmentos (SA, SB, SC, SD, SE, SF, SG), y que también funcionan a nivel bajo.

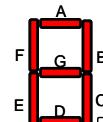


Figura 17.2:  
Segmentos del display

Recuerda que sólo tenemos 7 puertos para todos los segmentos y no  $7 \times 4 = 28$  segmentos. Por lo tanto, debes usar el mismo segmento para los cuatro *displays* (tal como lo hemos hecho en prácticas).

Las entradas y salidas del circuito se muestran en la figura 17.3. El reloj del circuito (clk) va a 10 MHz.

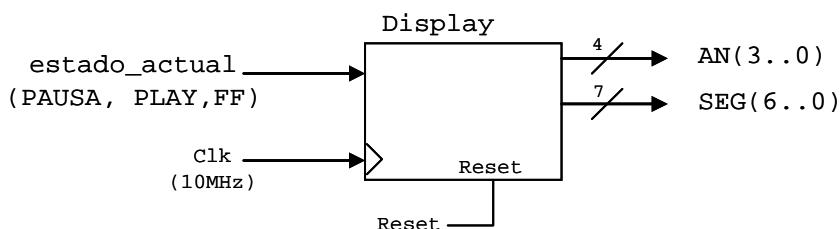


Figura 17.3: Esquema del circuito

**Se pide:**

- Realizar el esquema interno del circuito, indicando los bloques y explicando su funcionamiento.
- Realizar el modelo VHDL de la arquitectura circuito

## 17.2. Solución

La práctica del contador automático y la expendedora (apartado 12.2) son similares a este circuito. La explicación detallada de cómo hacer este circuito (multiplexar en el tiempo) se encuentra en la práctica del contador automático (apartado 6.3.2).

### 17.2.1. Esquema interno del circuito

El esquema del circuito se muestra en la figura 17.4

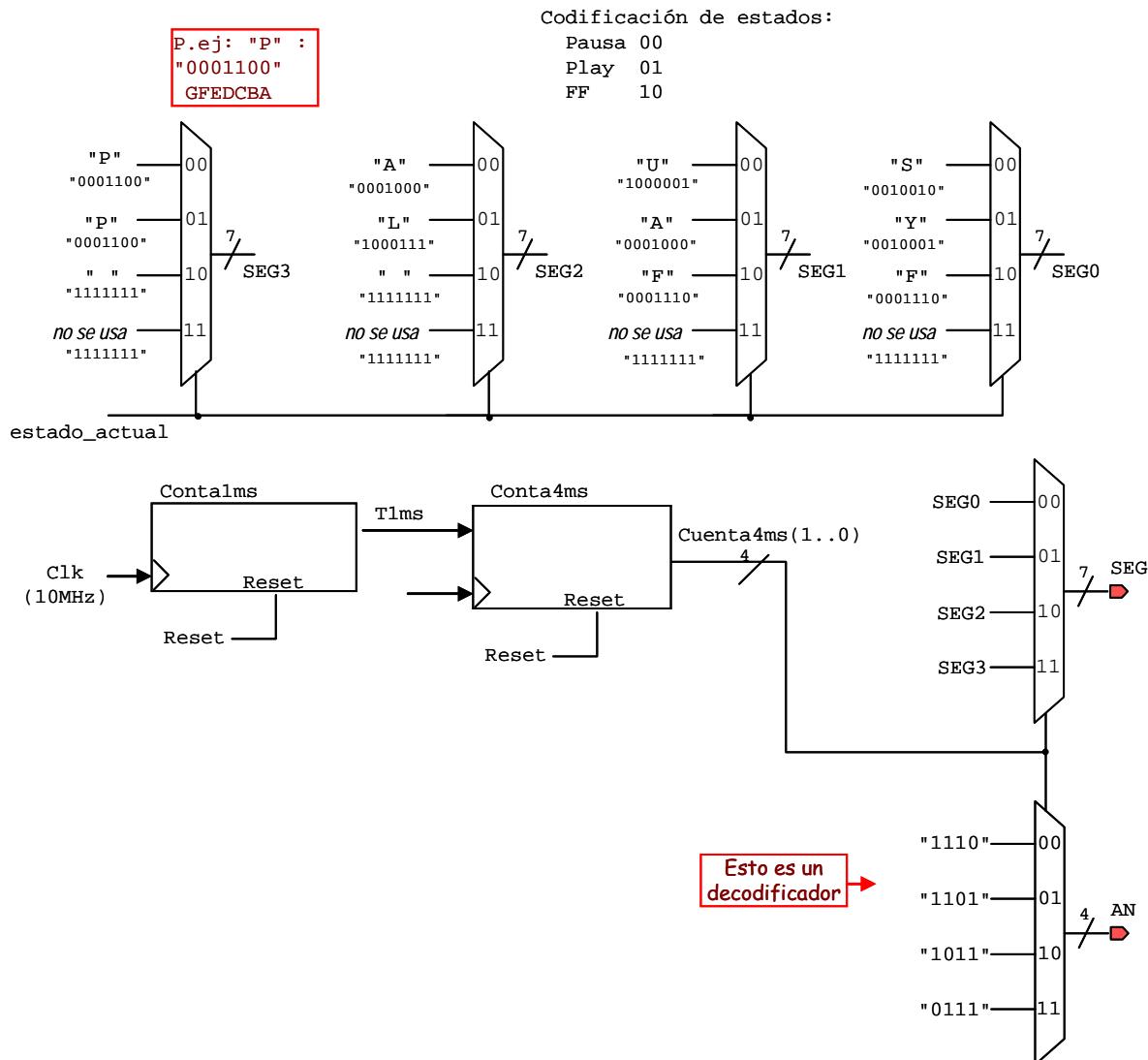


Figura 17.4: Esquema del circuito de visualización

### 17.2.2. Modelo en VHDL

A continuación se muestra el código VHDL que implementa el circuito de la figura 17.4.

```

architecture Behavioral of temp_mux is
    -- los 4 displays, preparados para ser mostrados segun la cuenta
    signal SEG0, SEG1, SEG2, SEG3 : std_logic_vector (6 downto 0);
    signal cuenta : unsigned (13 downto 0);      -- para contar hasta 10.000
    constant cfincuenta : natural := 10000;
    signal fin1mili : std_logic;
    signal cuenta4milis : unsigned (1 downto 0);
begin

```

```

--- proceso que segun el estado asigna a los 4 displays informacion diferente.
-- Si PAUSA muestra "PAUS" - Si PLAY muestra "PLAY" - Si FW muestra "FF"
P_QUE_MUESTRO: Process (estado_actual)
begin
  case estado_actual is
    when PAUSA =>
      --GFEDCBA
      SEG3 <= "0001100"; -- P
      SEG2 <= "0001000"; -- A
      SEG1 <= "1000001"; -- U
      SEG0 <= "0010010"; -- S
    when PLAY =>
      SEG3 <= "0001100"; -- P
      SEG2 <= "1000111"; -- L
      SEG1 <= "0001000"; -- A
      SEG0 <= "0010001"; -- Y
    when FW =>
      SEG3 <= "1111111"; -- nada
      SEG2 <= "1111111"; -- nada
      SEG1 <= "0001110"; -- F
      SEG0 <= "0001110"; -- F
  end case;
end process;
----- contamos un milisegundo, como el reloj va a 10 MHz, tenemos que contar 10000.
P_Cuenta_1mili: Process (Reset, Clk)
begin
  if Reset = '1' then
    cuenta <= (others => '0');
    fin1mili <= '0';
  elsif Clk'event and Clk='1' then
    if cuenta < cfincuenta then
      cuenta <= cuenta + 1;
      fin1mili <= '0';
    else
      cuenta <= (others => '0');
      fin1mili <= '1';
    end if;
  end if;
end process;
----- Ahora contamos cuatro cuentas de fin1mili
P_Cuenta_4milis: Process (Reset, Clk)
begin
  if Reset = '1' then
    cuenta4milis <= (others =>'0');
  elsif Clk'event and Clk='1' then
    if fin1mili = '1' then
      if cuenta4milis = 3 then
        cuenta4milis <= (others => '0');
      else
        cuenta4milis <= cuenta4milis + 1;
      end if;
    end if;
  end if;
end process;
----- ahora, gobernados por cuenta4milis, cada milisegundo mostramos un display
-- diferente. Y a la vez, damos la orden de activar el anodo correspondiente
P_Muestra_display: Process (cuenta4milis, SEG3, SEG2, SEG1, SEG0)
begin
  case cuenta4milis is
    when "00" =>
      SEG <= SEG0;
      AN <= "1110";
    when "01" =>
      SEG <= SEG1;
      AN <= "1101";
    when "10" =>
      SEG <= SEG2;
      AN <= "1011";
    when others =>
      SEG <= SEG3;
      AN <= "0111";
  end case;
end process;
end Behavioral;

```

Código 17-1: Arquitectura del circuito de visualización del MP3



## 18. Robot rastreador

Este problema se puso en el examen del 9 de septiembre de 2009.

### 18.1. Enunciado

Queremos realizar el control de un robot rastreador, esto es, un coche que sigue una línea negra sobre un suelo blanco. Para ello disponemos de un coche eléctrico de juguete y queremos realizar el circuito electrónico que controle el giro del coche (derecha o izquierda).

Para saber si el coche está sobre la línea negra o no, se ponen dos detectores de infrarrojos en la parte delantera del coche (ver figura de la derecha). Los detectores de infrarrojos están juntos y centrados en el alerón delantero del coche. El receptor que está a la izquierda se le llamará RI y el de la derecha RD.

Sin entrar a explicar el funcionamiento de los detectores de infrarrojos, lo único que nos importa para este problema es que los detectores dan un cero cuando están sobre la línea negra (ya que la luz emitida no se refleja) y devuelven un uno cuando están sobre el suelo blanco (porque se refleja la luz).

Así que usaremos las salidas de los detectores de infrarrojos como entradas de nuestro sistema de control. Por lo tanto, nuestro sistema de control tendrá dos entradas: RI y RD, que indican los valores devueltos por los receptores izquierdo y derecho respectivamente.

Como se muestra en la figura de la derecha, nuestro sistema tendrá dos salidas: GI y GD, que dan la orden de girar a la izquierda o derecha, respectivamente. Por ejemplo:  $GD = '1'$  será la orden de girar a la derecha. Si las dos están a cero significa que el coche debe ir recto. Obviamente, nunca se deberá dar la orden de girar a la derecha e izquierda simultáneamente.

El objetivo es dirigir el giro del coche de la siguiente manera:

1. Cuando los dos receptores estén sobre la línea negra, el coche está donde debe estar y por lo tanto no se deben girar las ruedas ( $GI = '0'$  y  $GD = '0'$ ). Ver figura 18.3.A.
2. Cuando uno de los receptores no esté sobre la línea negra (se recibe un '1' de dicho receptor), se debe girar de modo que se corrija la desviación. Esto es:
  - 2.1. Si el receptor de la izquierda no está sobre la línea negra pero el receptor de la derecha sí está, el coche se está saliendo por la izquierda y deberá girar a la derecha. Ver figura 18.3.B.

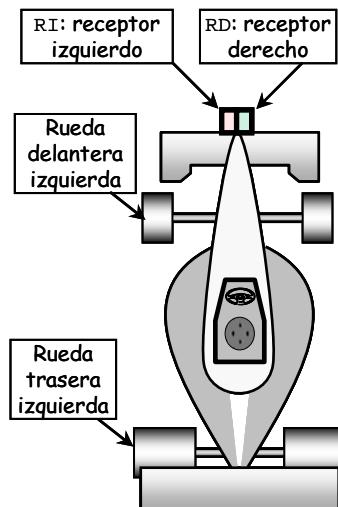


Figura 18.1: Coche visto desde arriba

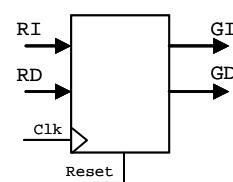


Figura 18.2: Esquema de entradas y salidas del sistema de control

- 2.2. Si el receptor de la derecha no está sobre la línea negra pero el receptor de la izquierda sí está, el coche se está saliendo por la derecha y deberá girar a la izquierda.
3. Cuando ninguno de los receptores estén sobre la línea negra, significa que el coche se ha salido completamente y deberá de corregir la desviación según por el lado por donde se haya salido. Ver figura 18.3.C

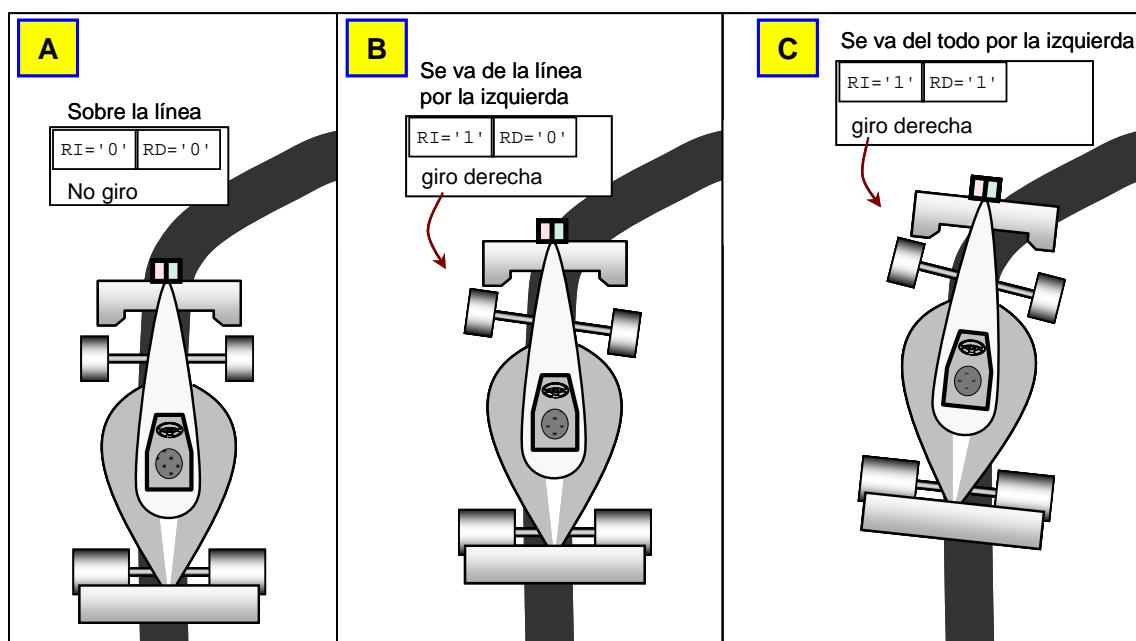


Figura 18.3: Distintas posiciones del coche respecto a la linea

Datos adicionales del sistema:

- Los dos receptores están lo más juntos posible y el ancho de la línea negra es mayor que el ancho de los dos receptores juntos. Por tanto nunca se podrá pasar de estar sobre solamente un receptor a estar sobre sólo el otro receptor. Esto es, entremedias siempre habrá un momento en el que se estará sobre los dos receptores
- El reloj del circuito (`clk`) es mucho más rápido que la mecánica del coche y su movimiento.
- Por esto último, nunca podremos pasar de estar con los dos receptores sobre la línea a estar totalmente fuera de la línea. Entremedias el coche estará con un receptor sobre la línea y el otro no.
- Existe un pulsador para resetear asíncronamente (`Reset`), pero no se debe de presionar durante el funcionamiento normal del circuito.
- Al empezar se pone el coche alineado. Esto es, los dos receptores estarán sobre la línea negra.
- La velocidad del coche va a ser constante y para esta versión del coche no la utilizaremos ni la controlaremos, así que no nos importa.

**Se pide:**

- Especificar las entradas, salidas y estados del sistema de control de giro del coche considerándolo como máquina de **Moore**. Realizar el diagrama de estados de la máquina de estados que controla el circuito.

- b) La tabla de estados siguientes y salidas
- c) La tabla de excitación de biestables para biestables J-K
- d) Ecuaciones simplificadas para las salidas y para las entradas J de dos de los biestables del circuito (no se piden las entradas K)

## 18.2. Solución

### 18.2.1. Entradas, salidas y estados. Diagrama de estados

Las entradas y salidas del sistema de control son las mismas que las de la figura 18.2 que se puso en el enunciado.

- **Dos entradas (RI, RD):**

- RI: receptor de infrarrojos de la izquierda.
- RD: receptor de infrarrojos de la derecha.

Para ambas, los valores son:

- '0': si está sobre la línea
- '1': si está fuera de la línea

Las cuatro combinaciones son posibles (ver figura 18.4)

- **Dos salidas (GI, GD)**

- GI='1': Orden para girar a la izquierda.
- GD='1': Orden para girar a la derecha.

Las posibles salidas son:

- (GI, GD)= (1, 0) → Giro a la izquierda
- (GI, GD)= (0, 1) → Giro a la derecha
- (GI, GD)= (0, 0) → Recto
- (GI, GD)= (1, 1) → No se debe poner nunca

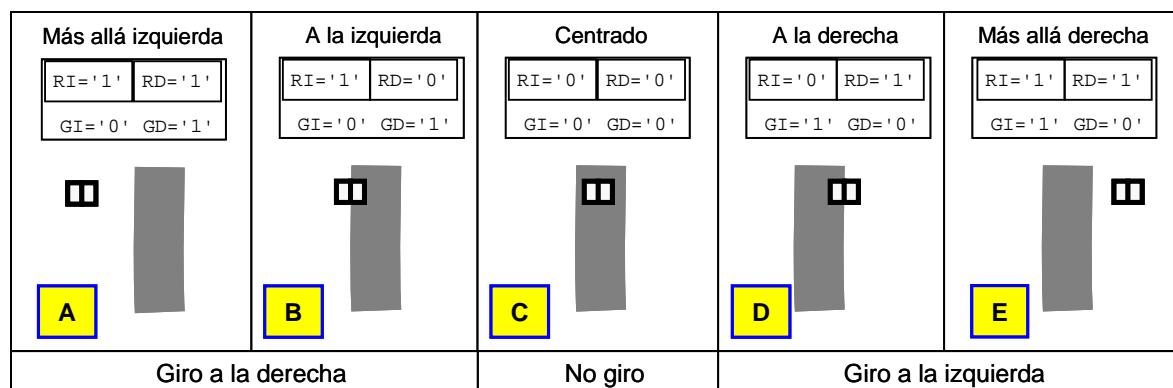


Figura 18.4: Posición de los receptores respecto a la línea y giro

Ahora hay que realizar el diagrama de transición de estados. Recordamos que en este apartado hay que realizarlo como una máquina de Moore.

El orden de las entradas y salidas en el diagrama será el siguiente (figura 18.5):

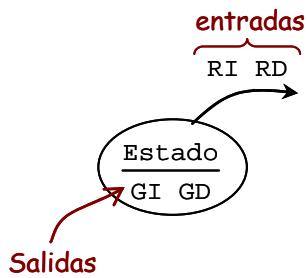


Figura 18.5: Orden de entradas y salidas en el diagrama de estados

Fíjate que es una máquina de Moore, por lo que las salidas están en el propio estado.

Inicialmente partimos del estado CENTRO, en donde  $RI = '0'$  y  $RD = '0'$ , ya que el enunciado dice que al principio se coloca el coche sobre la línea. En este estado se mantiene el coche sin girar, hasta que uno de los receptores cambie a 1.

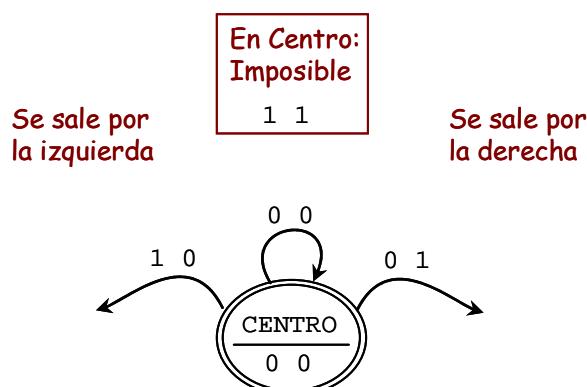


Figura 18.6: Estado inicial y sus transiciones

Estando en CENTRO, si el detector  $RI$  se pone a uno significa que nos salimos por la izquierda (ver figura 18.4.B). Y por lo tanto, hay que girar a la derecha. Lo contrario ocurre si el detector  $RD$  se pone a 1. Lo que no puede ocurrir nunca es que estando sobre la línea ambos receptores se pongan a la vez los dos a uno, pues primero se debe poner uno y luego el otro. Para ello creamos un estado IZQ al que vamos cuando desde CENTRO el receptor  $RI$  se pone a 1. En este estado, el giro se hace a la derecha. Saldremos de este estado cuando hayamos corregido la desviación y volvamos al centro (ambos receptores a 0). Lo mismo ocurre si el coche se desvía a la derecha.

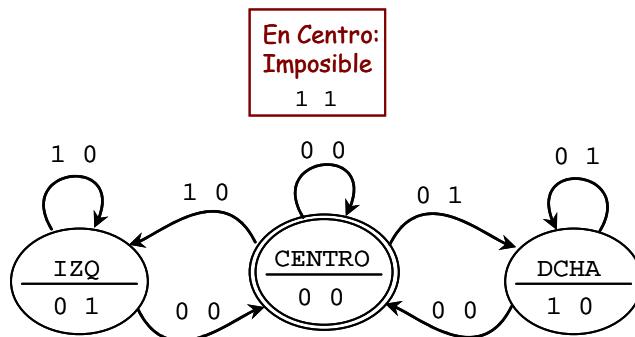


Figura 18.7: Estados con un detector sobre la línea y otro fuera

Sin embargo, puede ocurrir, que estando en los estados IZQ o DCHA el coche se salga totalmente de la línea. Eso lo sabremos porque ambos receptores están a 1. Para ello crearemos los estados FIZQ y FDCHA (de Fuera IZQuierda y Fuera DereCHA).

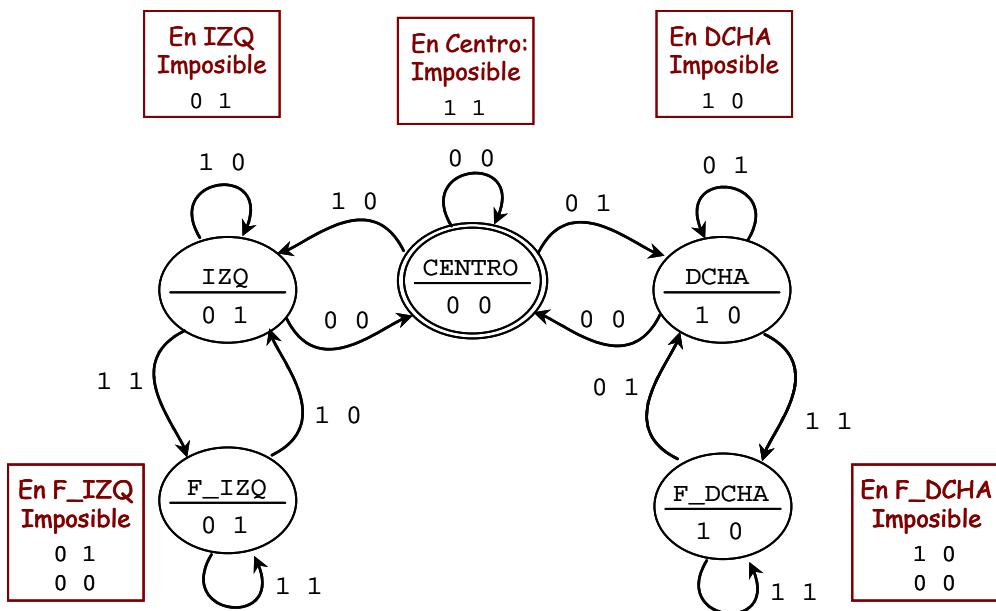


Figura 18.8: Diagrama de estados final sin reducir

El diagrama de la figura 18.8 podría ser el definitivo. Sin embargo, podemos fijarnos que en realidad los estados **F<sub>IZQ</sub>** y **F<sub>DCHA</sub>** no son necesarios ya que tienen la misma salida que **IZQ** y **DCHA** respectivamente, y en dichos estados, las mismas entradas conducen a los mismos estados (o son entradas imposibles). Así que se podría simplificar en el mostrado en la figura 18.9.

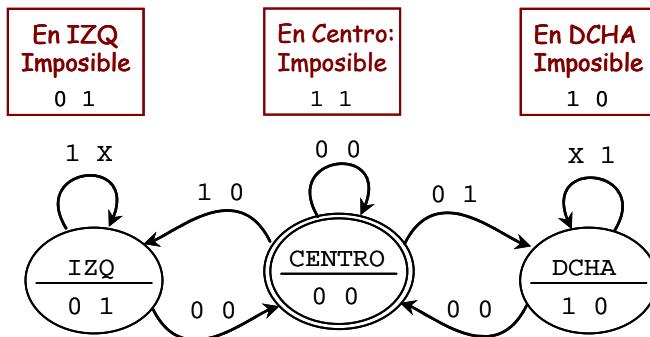


Figura 18.9: Diagrama de estados final reducido

Este diagrama no se puede simplificar más ya que cada estado tiene una salida distinta. Diseñándolo como máquina de Mealy sí se podría simplificar más (pero no se pedía en el examen).

### 18.2.2. Tabla de estados siguientes y salidas

Antes de nada tenemos que calcular el número de biestables que necesitamos y asignarles unos valores. Como tenemos 3 estados, nos vale con dos biestables. Asignaremos los mismos valores de los estados que de las salidas, pero vale cualquier otra combinación. La asignación se muestra en la tabla 18.1.

Estado	Q1	Q0
CENTRO	0	0
IZQ	0	1
DCHA	1	0

Tabla 18.1: Codificación de estados

La tabla de estados siguientes y salidas se muestra en la tabla 18.2.

**Moore: Sólo dependen del estado**

Estado actual	Entradas	Salidas	Estado siguiente
Q1 Q0	RI RD	GI GD	Q1 <sup>t+1</sup> Q0 <sup>t+1</sup>
0 0	0 0	0 0	0 0
0 0	0 1	0 0	1 0
0 0	1 0	0 0	0 1
0 0	1 1	0 0	x x
0 1	0 0	0 1	0 0
0 1	0 1	0 1	x x
0 1	1 0	0 1	0 1
0 1	1 1	0 1	0 1
1 0	0 0	1 0	0 0
1 0	0 1	1 0	1 0
1 0	1 0	1 0	x x
1 0	1 1	1 0	1 0
1 1	x x	x x	x x

CENTRO {

IZQ {

DCHA {

Sin usar (imposible)

→ condiciones imposibles

→ condiciones imposibles

→ condiciones imposibles

Tabla 18.2: Tabla de estados siguientes y salidas

### 18.2.3. Tabla de excitación de biestables para biestables J-K

La tabla de excitación de los biestables la podríamos haber rellenado en la tabla 18.2. Para saber las ecuaciones de las entradas de los biestables necesitamos saber qué necesitamos para lograr cada una de las transiciones. Como nos piden hacerlo con biestables J-K, ponemos su tabla, y a partir de ella obtenemos las entradas J y K para las cuatro posibles transiciones.

J	K	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	Q(t)

→

Q(t)	Q(t+1)	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Tabla 18.3: Tabla de entradas necesarias para obtener una transición en biestables J-K

Ahora nos fijamos en la tabla de los estados siguientes (tabla 18.2) para ver en cada caso qué transiciones hay y qué valores en las entradas de los biestables J-K necesitamos.

**Moore: Sólo dependen del estado**

Estado actual		Entradas		Salidas		Estado siguiente		Entradas en los biestables para conseguir las transiciones $Q_i \rightarrow Q_i^{t+1}$			
$Q_1$	$Q_0$	$RI$	$RD$	$GI$	$GD$	$Q_1^{t+1}$	$Q_0^{t+1}$	$J_1$	$K_1$	$J_0$	$K_0$
CENTRO	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 X	0 X	0 X	
	0 0	0 0	0 1	0 0	0 0	1 0	1 0	1 X	0 X	0 X	
	0 0	1 0	0 0	0 0	0 0	0 1	0 1	0 X	1 X	1 X	
	0 0	1 1	0 0	0 0	X X	X X	X X	X X	X X	X X	
IZQ	0 1	0 0	0 0	0 1	0 0	0 0	0 0	0 X	X 1	X 1	
	0 1	0 0	0 1	0 1	X X	X X	X X	X X	X X	X X	
	0 1	1 0	0 0	0 1	0 1	0 1	0 1	0 X	X 0	X 0	
	0 1	1 1	0 0	0 1	0 1	0 1	0 1	0 X	X 0	X 0	
DCHA	1 0	0 0	1 0	1 0	0 0	0 0	0 0	X 1	0 X	0 X	
	1 0	0 0	1 1	1 0	1 0	1 0	1 0	X 0	0 X	0 X	
	1 0	1 0	1 0	1 0	X X	X X	X X	X X	X X	X X	
	1 0	1 1	1 0	1 0	1 0	1 0	1 0	X 0	0 X	0 X	
Sin usar (imposible)	1 1	X X	X X	X X	X X	X X	X X	X X	X X	X X	

Tabla 18.4: Tabla de excitación de los necesarias para obtener una transición en biestables J-K

#### 18.2.4. Ecuaciones simplificadas

Las ecuaciones de las salidas son inmediatas porque las hemos hecho coincidir con los biestables de los estados. Así pues:

$$GI = Q_1$$

$$GD = Q_0$$

Para las entradas de los biestables utilizamos los mapas de Karnaugh. Se muestran en la figura 18.10, se incluyen las K, aunque no se pedían.

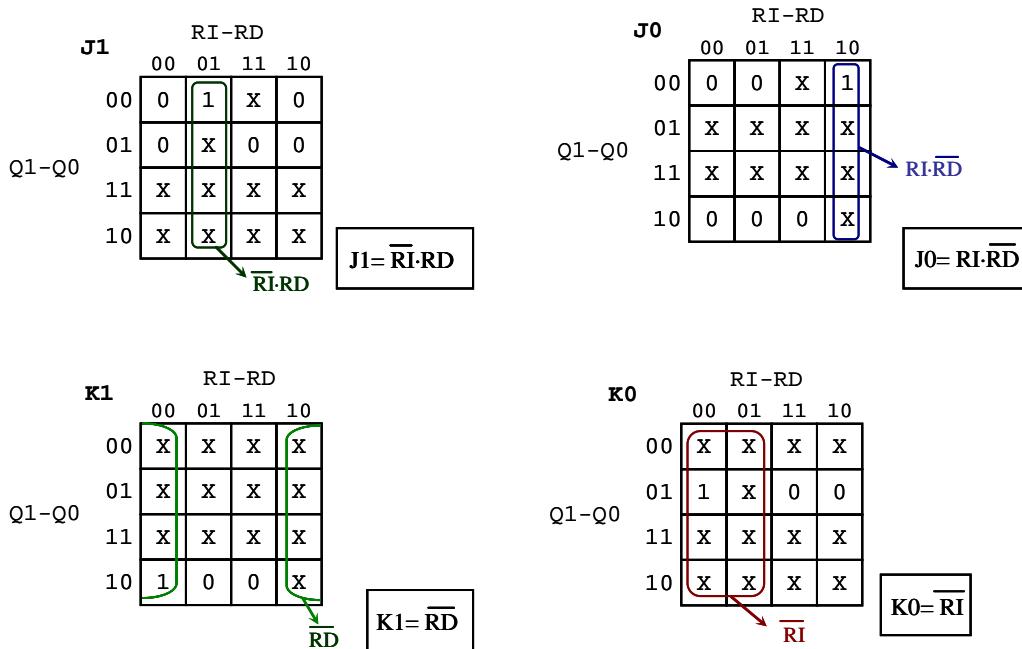


Figura 18.10: Mapas de Karnaugh y ecuaciones de las entradas de los biestables



## 19. Teclado de teléfono móvil

Este problema se puso en el examen del 9 de septiembre de 2009.

### 19.1. Enunciado

Se quiere realizar el circuito de una tecla de teléfono móvil que funcione como cuando se escriben mensajes de texto (sms). Para simplificar, se va a realizar el circuito para una única tecla y dicha tecla servirá para escribir los caracteres: A, B, 2 (para simplificar hemos eliminado la letra C y otros caracteres).

El funcionamiento resumido del circuito es el siguiente:

- Inicialmente no se muestra nada hasta que se pulsa la tecla  $P$ .
- Si quiero escribir la letra A, debo pulsar  $P$  una vez.
- Si quiero escribir la letra B, debo pulsar  $P$  dos veces rápidamente. En la pantalla se muestra A al pulsar la primera vez, y cuando vuelvo a pulsar se muestra B. Si tardase mucho en pulsar la segunda vez, se fijaría el valor de A (y se volvería a empezar para escribir en la siguiente posición).
- Si pulso  $P$  tres veces rápidamente obtengo el número 2 (recuerda que hemos quitado la letra C para simplificar). Cada vez que he pulsado  $P$ , se habrá mostrado el carácter correspondiente a la secuencia: A-B-2
- Si pulsase  $P$  cuatro veces rápidamente obtendría de nuevo la letra A. Ya que la secuencia es circular: A-B-2-A-B-2-A-.... Obteniendo el carácter correspondiente según el número de veces que pulse.
- Si después de haber pulsado  $P$ , no vuelvo a pulsar  $P$  durante un intervalo de un segundo o más, se fija el último carácter que se tenga, y a continuación se vuelve al estado inicial (para escribir en la siguiente posición).
- En cualquier estado, si mantengo  $P$  pulsado durante un segundo o más, se fija el valor numérico (2) y al soltar  $P$  se vuelve al estado inicial (para escribir en la siguiente posición).

A continuación se muestran las entradas y salidas del circuito:

**Entradas** (además del reloj y reset):

- $P$ : Pulsador de la tecla del móvil (recuerda que en este móvil para simplificar sólo tenemos una tecla)

**Salidas:**

- **A**: Mostrar el carácter A
- **B**: Mostrar el carácter B
- **DOS**: Mostrar el número 2
- **LISTO**: Indica que el carácter indicado por una de las salidas (A, B, DOS) está listo para fijarlo, y se deberá pasar a la siguiente posición. Sólo está activa durante un ciclo de reloj.

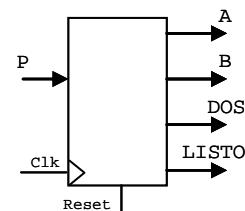


Figura 19.1: Esquema de entradas y salidas del sistema de control

El funcionamiento detallado con tiempos es el siguiente:

- Inicialmente no se muestra ningún carácter (las cuatro salidas a cero), en este estado se espera a que se pulse  $P$ .
- Si desde el estado inicial se pulsa  $P$  manteniéndola presionada durante menos de un segundo, se muestra el carácter  $A$  ( $A='1'$ ). La salida  $A$  se pone a uno desde que se comienza a pulsar  $P$ , las otras salidas  $B$ ,  $DOS$ ,  $LISTO$  se quedan a cero.
- Si una vez que he pulsado  $P$ , no vuelvo a pulsar  $P$ , transcurrido un segundo el valor se fija. Esto es:  $LISTO='1'$  y se mantiene la última salida que estaba a uno:  $A$ ,  $B$  ó  $DOS$ .
- En el caso anterior, la salida  $LISTO$  sólo se debe mantener activa durante un ciclo de reloj, tras lo cual el sistema vuelve al estado inicial, donde todas las salidas se ponen a cero ( $A$ ,  $B$ ,  $DOS$  y  $LISTO$ ).
- Sin embargo, si he pulsado  $P$ , (haciendo que, por ejemplo, se haya activado  $A$ ,  $A='1'$ ), y vuelvo a pulsar  $P$  antes de que transcurra un segundo, se activará la siguiente salida en la secuencia (por ejemplo, si  $A$  estaba a uno; se pondrá  $A='0'$  y  $B='1'$ ). El resto de salidas se mantienen a cero.
- En cualquier estado, si pulso  $P$  durante un segundo o más, se fija el 2:  $DOS='1'$  y  $LISTO='1'$ . Esto ocurre durante un ciclo de reloj, luego se ponen todas las salidas a cero, y no se hace nada hasta que se deje de pulsar  $P$ . Al dejar de pulsar  $P$  se vuelve al estado inicial.

#### Consideraciones:

- La señal de reloj va a 1MHz
- La señal de reset es activa a nivel alto y no se usa en el funcionamiento normal del circuito
- De las salidas de caracteres:  $A$ ,  $B$  y  $DOS$ , sólo puede haber una de ellas activa simultáneamente. Sí puede ser que ninguna de ellas esté activa.
- La entrada  $P$  no tiene rebotes. Por tanto, con la frecuencia del reloj que se tiene, será imposible pulsar la tecla  $P$  durante un sólo ciclo de reloj, los pulsos serán bastante más largos.
- Todas las entradas y salidas del circuito son de un bit, no se pide mostrar nada en displays de siete segmentos.

#### Se pide:

- a) Dibujar los bloques internos del circuito, con entradas, salidas y las señales intermedias que se vayan a usar. Indicar la función de cada uno de los bloques internos.
- b) En correspondencia con lo anterior, dibujar el diagrama de estados de la máquina de Mealy que controla el circuito.

No se pide realizar la tabla de estados.

## 19.2. Solución

### 19.2.1. Bloques internos del circuito

Para realizar este circuito necesitamos una máquina de estados y un temporizador que nos permitirá medir si ha transcurrido un segundo después de pulsar la tecla  $P$  y si se mantiene la tecla presionada durante más de un segundo.

El circuito tiene dos bloques: la máquina de estados y un temporizador.

De la máquina de estados sale una señal intermedia  $E$  que va al temporizador. Esta señal funciona como habilitación (*enable*) de modo que cuando  $E$  es cero, el contador se pone a cero y no cuenta, mientras que cuando  $E$  es uno, se habilita la cuenta.

La señal  $T$  indica cuándo ha transcurrido un segundo desde que  $E$  se ha mantenido activo ( $E = '1'$ ).

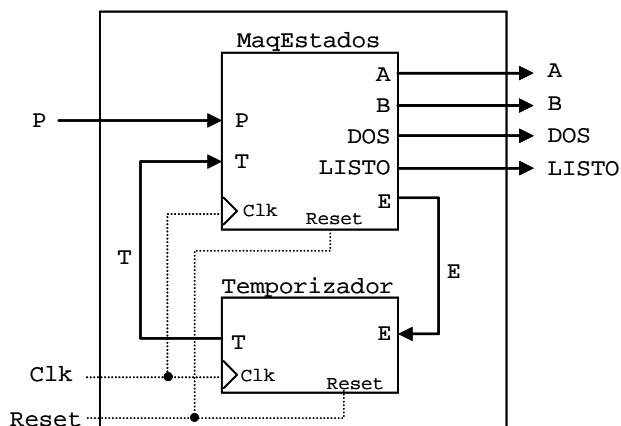


Figura 19.2: Bloques internos del circuito

### 19.2.2. Diagrama de estados

Para el diagrama de estados usaremos el siguiente orden de entradas y salidas

<b>entradas</b> $P \ T \ / \ A \ B \ DOS \ LISTO \ E$ <b>Salidas</b> 	<b>entradas</b> $P \ T \ / \ A \ B \ D \ L \ E$ <b>Salidas</b> 
<i>Figura 19.3: Orden de entradas y salidas en el diagrama de estados</i> <i>Figura 19.4: Orden de entradas y salidas en el diagrama de estados. Salidas con iniciales para simplificar</i>	

Como hay muchas salidas, para que el diagrama sea más fácil de entender, en vez de poner unos, se van a poner las iniciales de las señales (figura 19.4). Los ceros se pondrán como ceros. Fíjate que es una máquina de Mealy.

Empezamos a hacer del diagrama según las instrucciones:

Partimos del estado **INICIAL**, donde esperamos a que se pulse  $P$ , todas las salidas están a cero, incluida la habilitación del contador, porque ahora no hay que contar segundos (el usuario puede tardar lo que quiera en escribir un carácter).

Como la habilitación del temporizador está a cero ( $E = '0'$ ), será imposible que  $T$  sea uno.

Nos quedamos en el estado **INICIAL** hasta que se pulse  $P$ , cuando esto ocurre tenemos que activar la salida  $A$  y el temporizador ( $E = '1'$ ).

Cuando se ha pulsado  $P$ , tenemos que ver durante cuánto tiempo se pulsa  $P$ :

**En INICIAL**  
**Imposible**  
 $T = '1'$   
 $X \ 1$

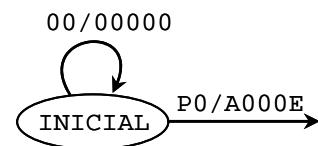


Figura 19.5: Estado inicial

- Si se pulsa durante un segundo o más, indica que tenemos que fijar el número 2.
- Si se pulsa menos de un segundo, hemos elegido la letra A, pero tenemos que seguir haciendo comprobaciones. Esto ocurre cuando las entradas son  $(P, T)$ , esto es, la tecla sigue presionada y acaba de llegar la señal de que ha pasado un segundo ( $T$ ).
- Si no se cumplen ninguna de estas dos, seguimos esperando, y las entradas para esto son  $(P, 0)$

Para comprobar esto, creamos un estado `ESP_S_A` (ESPer a Soltar, A) que indica que estamos esperando a que se suelte la tecla y que recuerda que la letra que se está mostrando es la A.

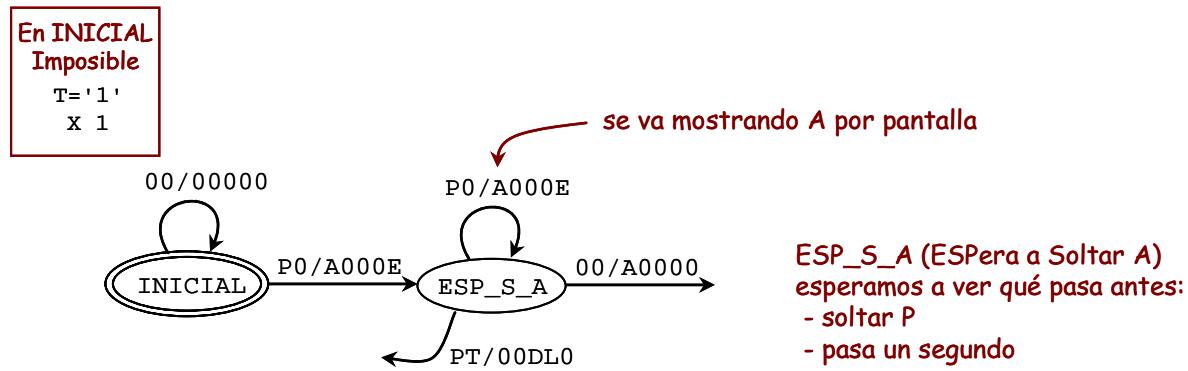


Figura 19.6: Estado de espera

Cuando se fija el número (se presiona  $T$  durante un segundo o más), la salida `LISTO`, se debe poner a uno durante un ciclo de reloj y luego se debe esperar a que se suelte la tecla (estado `ESP_S`). Si no hacemos esta espera, en el estado `INICIAL` consideraríamos que se ha vuelto a presionar  $P$ , y eso sería un error porque iríamos de nuevo al estado `ESP_S_A`.

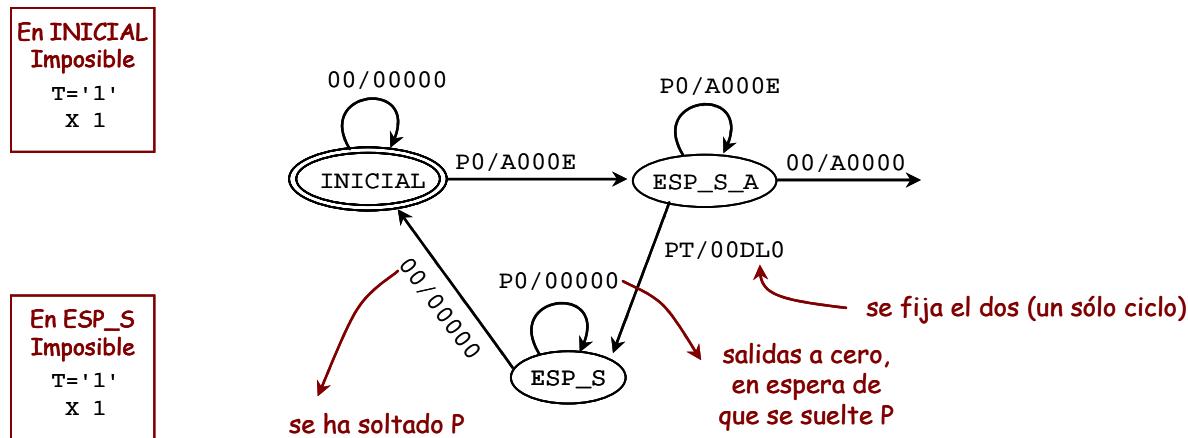


Figura 19.7: Estado de espera a que suelte la tecla

En el enunciado indica que si se pulsa  $T$  por un segundo o más, se debe de fijar el 2. Esto incluye el caso improbable (aunque no imposible) en el que soltamos la tecla justo en el instante en el que llega el aviso de que ha pasado un segundo. En este caso debemos fijar el 2 (lo indica el enunciado). La entrada para esto sería  $(0, T)$ .

En este caso no haría falta ir al estado `ESP_S` sino que se iría directamente al estado `INICIAL`. De todos modos, ir al estado `ESP_S` no sería un error, porque inmediatamente en el ciclo de reloj siguiente se iría al estado `INICIAL`, y es imposible que el usuario consiga

presionar el pulsador con solo un ciclo de reloj entre medias (y las teclas no tienen rebotes).

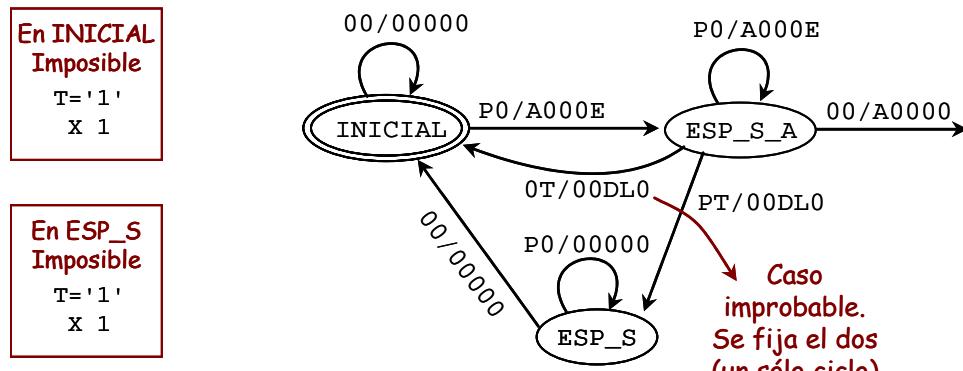


Figura 19.8: Se fija el dos en el caso improbable que suelta la tecla a la vez que termina la temporización

Ahora, estando en el estado `ESP_S_A`, esto es, el pulsador presionado y mostrando la letra A. Si se suelta la tecla antes de que pase un segundo, entradas: (0, 0). Tenemos que:

- Esperar a ver si se vuelve a presionar P antes de un segundo → estado `ESP_P_A`
- Para contabilizar los segundos, tenemos que reiniciar el contador poniendo su habilitación a cero durante el cambio de estado ( $E='0'$ ).

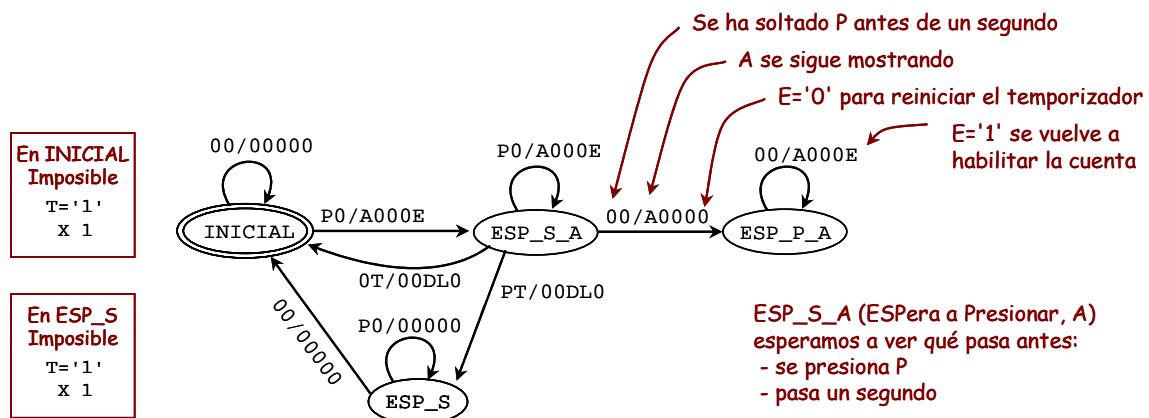


Figura 19.9: Se suelta la tecla antes de que pase un segundo: nueva espera

Ahora, estando en el estado `ESP_P_A`, esto es, el pulsador sin presionar y mostrando la letra A, puede pasar:

- Que antes de que pase un segundo se vuelva a presionar P. En este caso volvemos al estado a `ESP_S_B` (similar a `ESP_S_A`)
- Que pase un segundo sin volver a presionar P, entradas (0, T) → en este caso se fija la letra A
- En el caso improbable que ambos eventos ocurran simultáneamente, el enunciado dice que se fije la letra A. Las entradas serían (P, T), para este caso, podemos irnos al estado inicial (como en el caso anterior, y de este nos iríamos en el siguiente ciclo de reloj a `ESP_S_A` (pues P va a seguir presionada ya que no la podremos pulsar durante un único ciclo de reloj), o nos podemos ir directamente al estado `ESP_S_A` (lo que supone poner una flecha más). En el diagrama hemos elegido la primera opción (ambas son válidas).

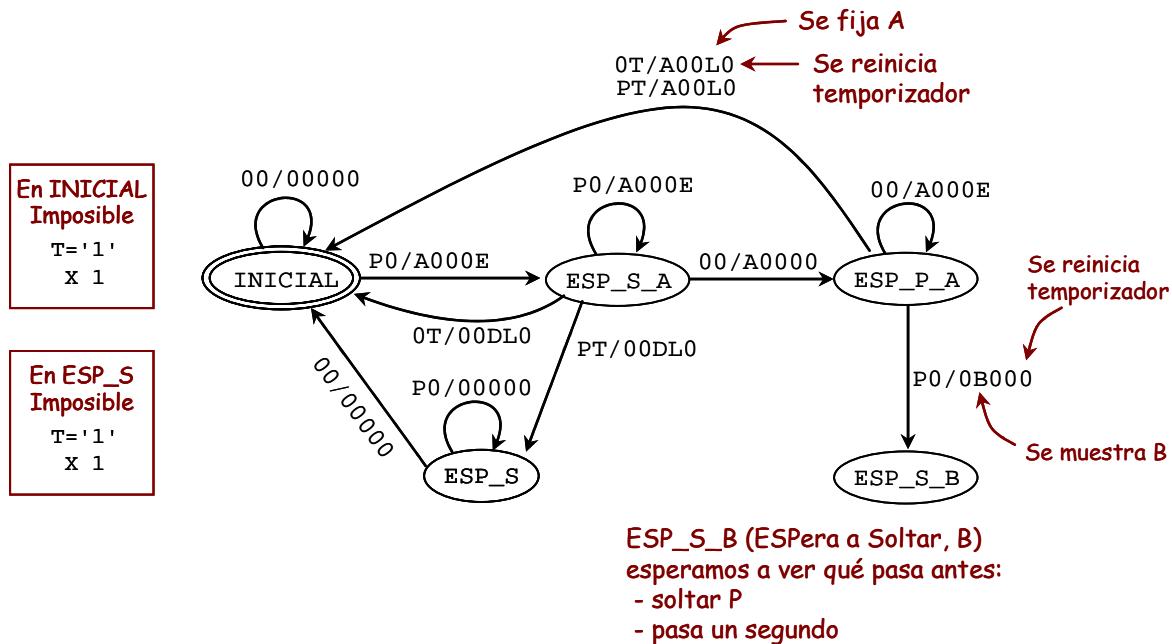


Figura 19.10: Nueva espera a que se suelte B

Ahora, estando en el estado `ESP_S_B`, esto es, el pulsador presionado y mostrando la letra B, puede pasar:

- ( $P, 0$ ): No ha pasado un segundo y sigue P pulsada → Seguimos en `ESP_S_B`
- $(0, 0)$ : Se suelta P antes de un segundo → Esperamos a ver si se vuelve a presionar P antes de un segundo
- $(P, T)$ : Ha pasado un segundo y no hemos soltado P → Fijamos el 2, y esperamos a que suelte el pulsador: `ESP_S`
- $(0, T)$ : Caso improbable de soltar justo en el segundo, lo tratamos como el anterior, pero nos vamos al estado `INICIAL` (si fuésemos al estado `ESP_S`, sería válido porque el circuito funcionaría igual desde el punto de vista del usuario)

Recuerda que para contabilizar los segundos, tenemos que reiniciar el contador poniendo su habilitación a cero durante el cambio de estado ( $E='0'$ ), y luego ponerla a uno.

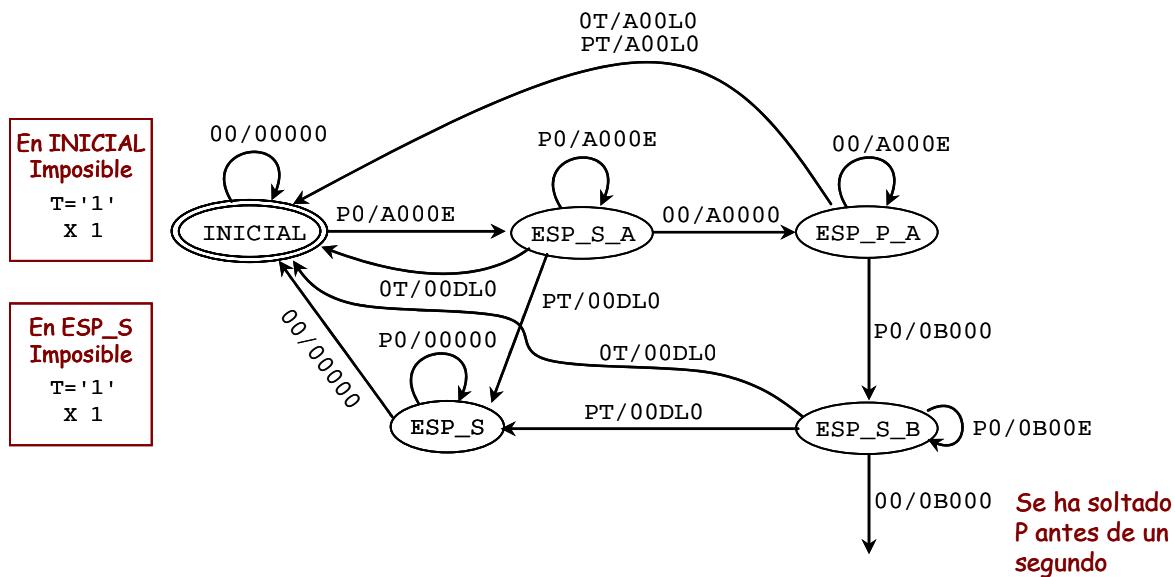


Figura 19.11: Se ha soltado P antes de que pase un segundo

Ahora, estando en el estado `ESP_P_B`, esto es, el pulsador sin pulsar y mostrando la letra B, puede pasar:

- ( $P, 0$ ): Se ha pulsado antes de que pase un segundo: Se muestra el dos  $\rightarrow$  `ESP_S_2`
- ( $0, 0$ ): No ha pasado un segundo y no se ha pulsado P  $\rightarrow$  Seguimos esperando
- ( $0, T$ ): Ha pasado un segundo y pulsado P  $\rightarrow$  Fijamos la B, y vamos al estado: `INICIAL`
- ( $P, T$ ): Caso improbable de pulsar justo en el segundo, lo tratamos como el anterior (también podríamos irnos a `ESP_S_A`).

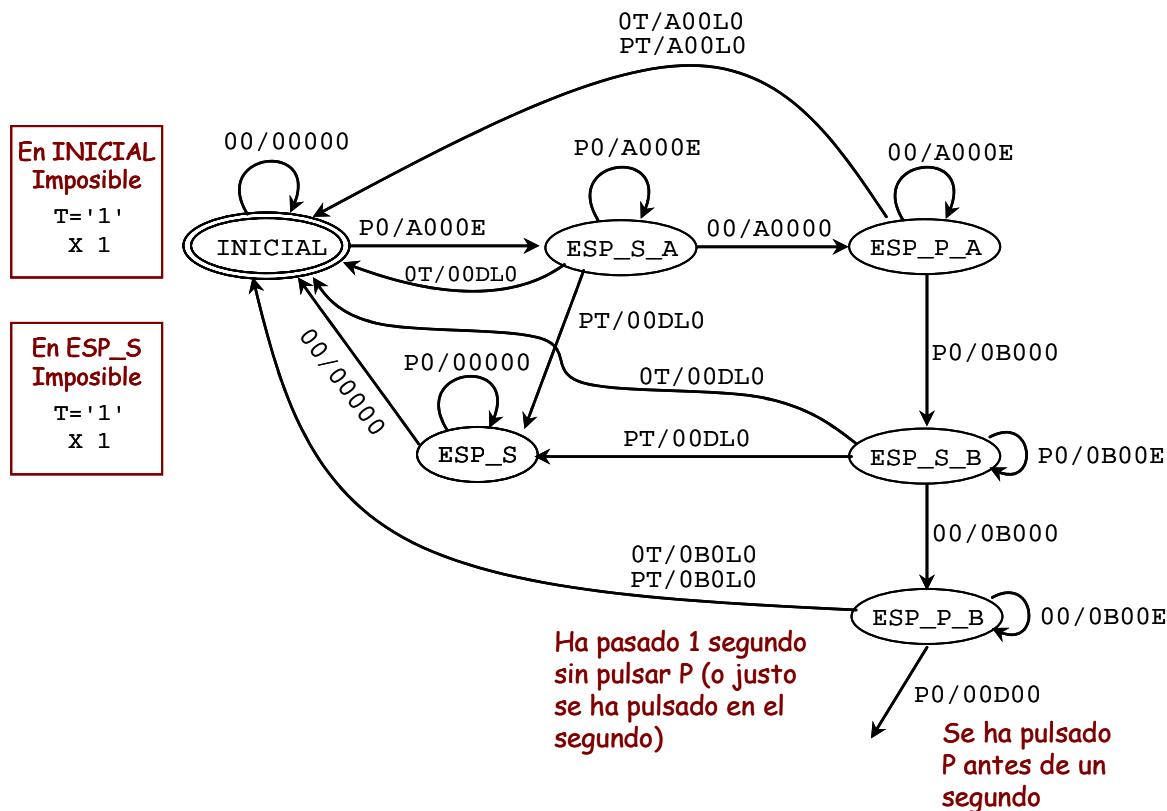


Figura 19.12: En la espera a ver si se vuelve a pulsar P mostrando la B

Y los dos estados que quedan se hacen de manera similar.

Sólo hay que tener en cuenta que después del 2 viene la A. Y por eso, del estado `ESP_P_2` viene `ESP_S_A`.

El diagrama de estados final se muestra en la figura 19.13.

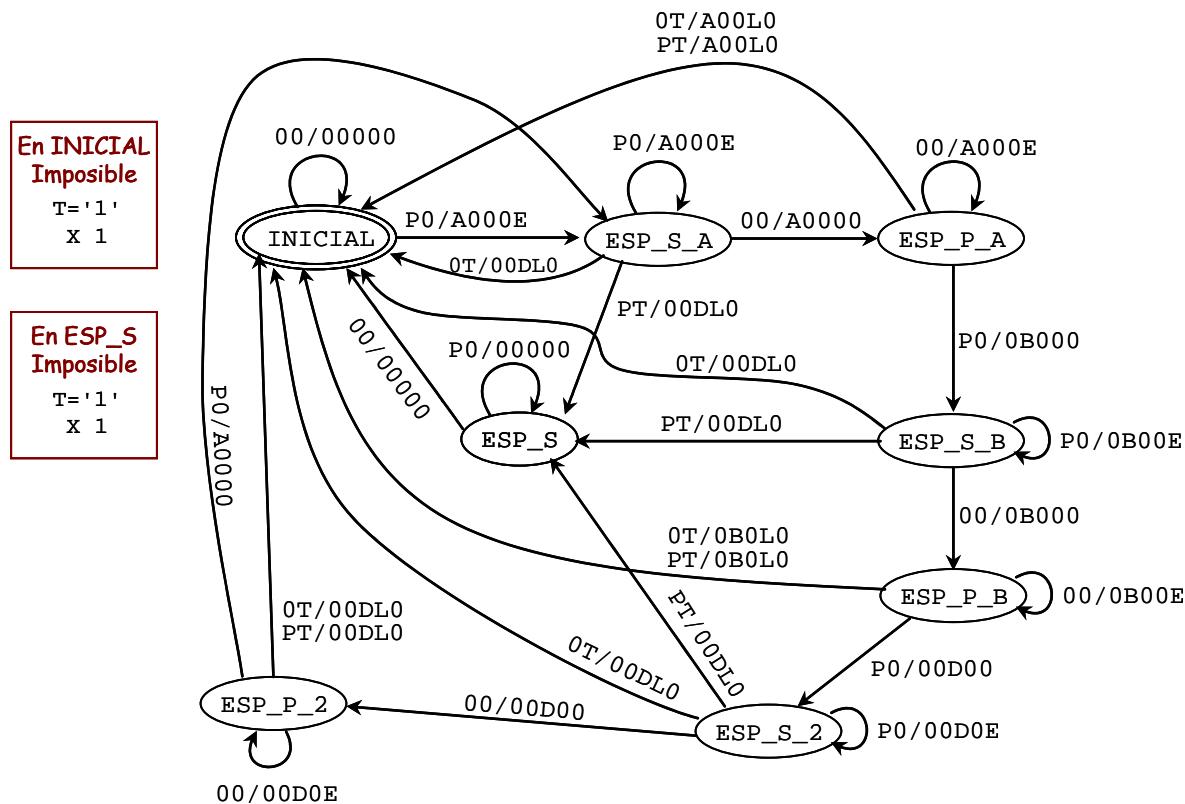


Figura 19.13: Diagrama de estados final

## 20. Cálculo de temporización de un circuito 1

Este problema se puso en el examen del 30 de enero de 2009.

### 20.1. Enunciado

1. Explicar brevemente qué es la metaestabilidad y por qué se produce
2. Calcular la frecuencia máxima a la que puede funcionar el siguiente circuito

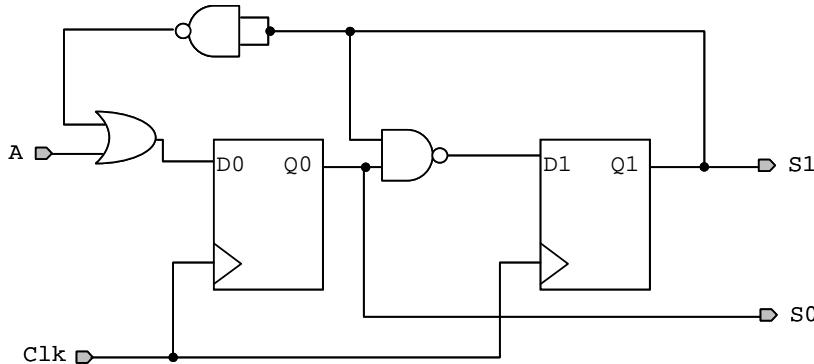


Figura 20.1: Circuito para analizar

Teniendo en cuenta las siguientes características:

	Tiempo de propagación	Tiempo de set-up	Tiempo de hold	Frecuencia máxima
Biestables	$t_{p\ max} = 20\ \text{ns}$ $t_{p\ min} = 4\ \text{ns}$	$t_{su} = 10\ \text{ns}$	$t_h = 5\ \text{ns}$	$f_{max} = 25\ \text{MHz}$
Puertas lógicas	$t_{p\ max} = 10\ \text{ns}$ $t_{p\ min} = 5\ \text{ns}$	----	----	----
Entradas	$t_{p\ max} = 10\ \text{ns}$ $t_{p\ min} = 5\ \text{ns}$			

Tabla 20.1: Características de los componentes del circuito

### 20.2. Solución

#### 20.2.1. Metaestabilidad

Metaestabilidad es un estado inestable de un biestable, en el que el biestable no está a un valor '0' ó '1'. Este estado se produce por no cumplirse los parámetros temporales del biestable. Cuando no se respetan los tiempos de *setup* y de *hold* del biestable

#### 20.2.2. Frecuencia máxima

Veamos las entradas de los biestables (D0 y D1)

entradas → salidas ↓	D0	D1
A	$t_{p\ max}(A) + t_{p\ max}(\text{puerta}) + t_{su}(D0) = 30\text{ns}$	--
Q0	--	$t_{p\ max}(D0) + t_{p\ max}(\text{puerta}) + t_{su}(D1) = 40\text{ ns}$ $t_{p\ max}(D1) + t_{p\ max}(\text{puerta}) + t_{su}(D1) = 40\text{ ns}$
Q1	$t_{p\ max}(D1) + 2 \cdot t_{p\ max}(\text{puerta}) + t_{su}(D0) = 50\text{ ns}$	--

Tabla 20.2: Tiempos de propagación máximos

El camino crítico tarda 50 ns, por lo tanto la frecuencia máxima es:

$$\frac{1}{50\text{ ns}} = \frac{10^9}{50}\text{ Hz} = \frac{1000}{50} \cdot 10^6\text{ Hz} = 20\text{ MHz}$$

Como es una frecuencia menor que la frecuencia máxima de los biestables (25 MHz), la frecuencia máxima de este circuito será de **20 MHz**.

## 21. Análisis de un circuito 1

Este problema se puso en el examen del 30 de enero de 2009.

### 21.1. Enunciado

Del circuito de capítulo 20 (figura 20.1):

- 1) Obtener la tabla de excitación del autómata
- 2) Dibujar el diagrama de transición de estados
- 3) ¿Es una máquina de Moore o de Mealy? ¿Por qué?

### 21.2. Solución

#### 21.2.1. Tabla de excitación del autómata

Primero obtenemos las ecuaciones de las entradas de los biestables y las salidas.

Ecuaciones

$$D_0 = A + \overline{Q_1}$$

$$S_0 = Q_0$$

$$D_1 = \overline{Q_0} \cdot Q_1$$

$$S_1 = Q_1$$

A partir de las ecuaciones obtenemos el estado siguiente. Como son biestables D, el valor de Q es directamente el del dato del biestable (D)

Estados		Entrada A	Estado siguiente	
Q <sub>1</sub>	Q <sub>0</sub>		Q <sub>1</sub> '	Q <sub>0</sub> '
0	0	0	1	1
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	0	0
1	1	1	0	1

Tabla 21.1: Tabla del estado siguiente

Organizamos la tabla de manera que sea más fácil ver el estado siguiente según la entrada.

Estados	Estado siguiente	
	A=0	A=1
Q <sub>1</sub> Q <sub>0</sub>	Q <sub>1</sub> ' Q <sub>0</sub> '	Q <sub>1</sub> ' Q <sub>0</sub> '
0 0	1 1	1 1
0 1	1 1	1 1
1 0	1 0	1 1
1 1	0 0	0 1

Tabla 21.2: Tabla de excitación del autómata

### 21.2.2. Diagrama de transición de estados

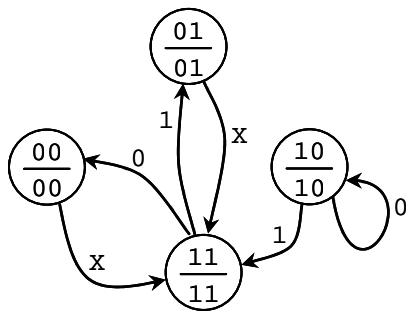


Figura 21.1: Diagrama de transición de estados

El último apartado: es una máquina de Moore porque las salidas no dependen de las entradas. Esto se puede deducir directamente observando el circuito.

## 22. Cálculo de temporización de un circuito 2

Este problema se puso en el examen del 11 de septiembre de 2009.

### 22.1. Enunciado

Calcular la frecuencia máxima a la que puede funcionar el siguiente circuito

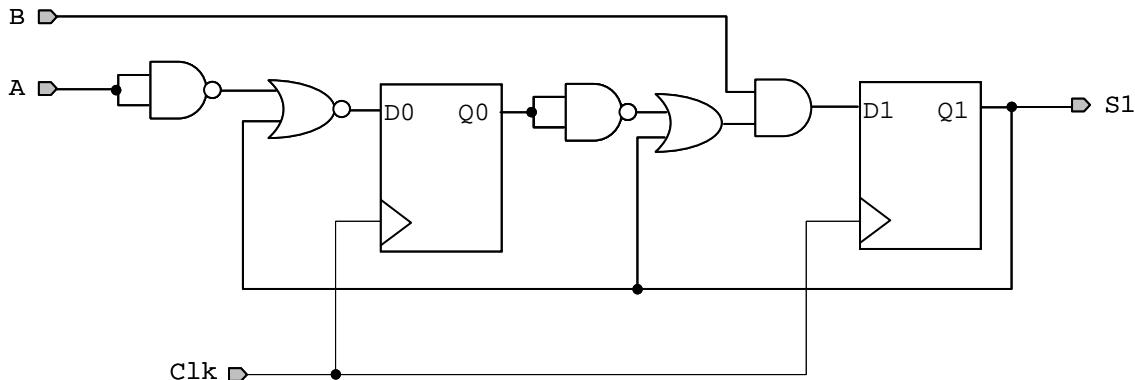


Figura 22.1: Circuito para analizar

Teniendo en cuenta las siguientes características:

	Tiempo de propagación	Tiempo de set-up	Tiempo de hold	Frecuencia máxima
Biestables	$t_p \max = 25 \text{ ns}$ $t_p \min = 11 \text{ ns}$	$t_{su} = 20 \text{ ns}$	$t_h = 10 \text{ ns}$	$f_{\max} = 20 \text{ MHz}$
Puertas lógicas	$t_p \max = 15 \text{ ns}$ $t_p \min = 7 \text{ ns}$	----	----	----
Entradas	$t_p \max = 50 \text{ ns}$ $t_p \min = 16 \text{ ns}$			

Tabla 22.1: Características de los componentes del circuito

### 22.2. Solución

Veamos cuál es el camino crítico:

Salidas Entradas	D0	D1
A	$t_p \max(A) + 2 \cdot t_p \max(\text{puerta}) + t_{su}(D0) = 50 + 2 \cdot 15 + 20 = 100 \text{ ns}$	
B		$t_p \max(B) + t_p \max(\text{puerta}) + t_{su}(D1) = 50 + 15 + 20 = 85 \text{ ns}$
Q0		$t_p \max(Q0) + 3 \cdot t_p \max(\text{puerta}) + t_{su}(D0) = 25 + 3 \cdot 15 + 20 = 90 \text{ ns}$
Q1	$t_p \max(Q0) + t_p \max(\text{puerta}) + t_{su}(D0) = 25 + 15 + 20 = 60 \text{ ns}$	$t_p \max(Q1) + 2 \cdot t_p \max(\text{puerta}) + t_{su}(D1) = 25 + 2 \cdot 15 + 20 = 75 \text{ ns}$

Tabla 22.2: Tiempos de propagación máximos

El camino crítico va de la entrada A al biestable D0. Como son 100ns, la frecuencia máxima es

$$\frac{1}{100\text{ns}} = \frac{10^9}{100} \text{ Hz} = 10 \text{ MHz}$$

Como 10 MHz es una frecuencia menor que la máxima fijada por los biestables, la frecuencia máxima del circuito será 10 MHz.

## 23. 24. Análisis de un circuito 2

Este problema se puso en el examen del 11 de septiembre de 2009.

### 24.1. Enunciado

Del circuito siguiente:

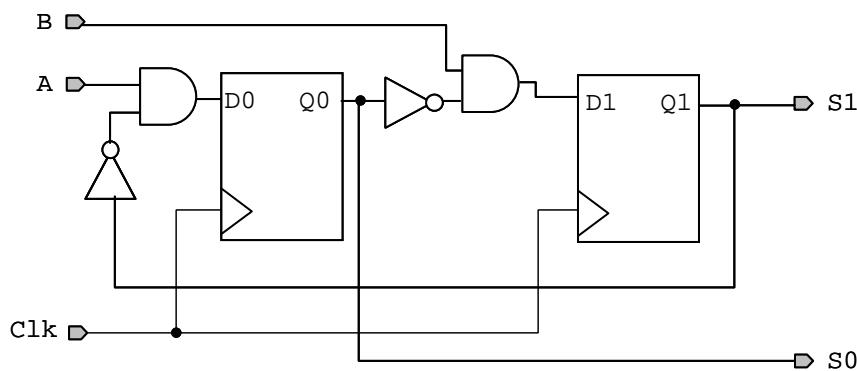


Figura 24.1: Circuito para analizar

- 1) Obtener la tabla de excitación del autómata
- 2) El diagrama de transición de estados
- 3) ¿Es una máquina de Moore o de Mealy? ¿Por qué?

### 24.2. Solución

#### 24.2.1. Tabla de excitación del autómata

Primero obtenemos las ecuaciones de las entradas de los biestables y las salidas. Incluimos la tablas de verdad para que luego nos sea más fácil llenar la tabla del autómata.

$$\begin{aligned} D0 &= A \cdot \overline{Q1} \\ D1 &= B \cdot \overline{Q0} \end{aligned} \quad \begin{aligned} S0 &= Q0 \\ S1 &= Q1 \end{aligned}$$

$Q1$	$A$	$D0$	$Q0$	$B$	$D1$
0	0	0	0	0	0
0	1	1	0	1	1
1	0	0	1	0	0
1	1	0	1	1	0

Tabla 24.1: Tablas de verdad de las entradas de los biestables

A partir de las ecuaciones obtenemos los estados siguientes según las entradas:

Q1	Q0	A	B	Q1'	Q0'
0	0	0	0	0	0
0	0	0	1	1	0
0	0	1	0	0	1
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	0	1	1	0
1	0	1	0	0	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

Organizamos la tabla para poder ver mejor los cambios de estados con las entradas

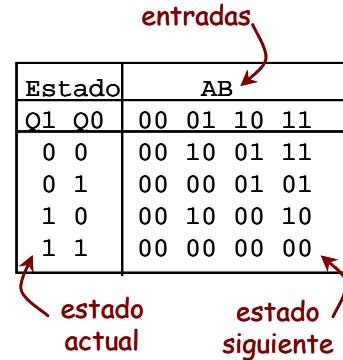
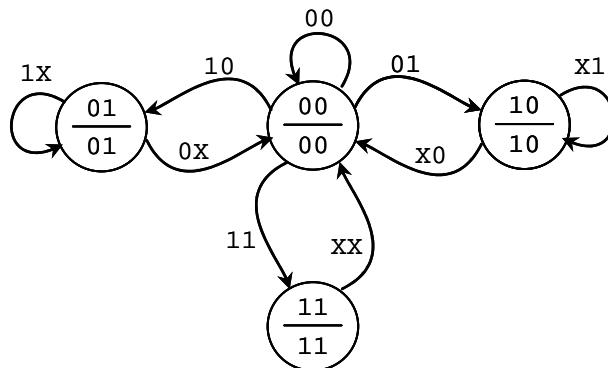


Tabla 24.3: Tabla de excitación del autómata

*Tabla 24.2: Tabla del estado siguiente*

#### **24.2.2. Diagrama de transición de estados**

Con cualquiera de estas tablas obtenemos los estados siguientes para dibujar el diagrama de estados:



*Figura 24.2: Diagrama de transición de estados*

El último apartado: es una máquina de Moore porque las salidas no dependen de las entradas. Esto se puede deducir directamente observando el circuito).

## Referencias

- [1] *Adept* de *Digilent*. Programa gratuito para programar las FPGAs por USB:  
<http://www.digilentinc.com/Products/Detail.cfm?Prod=ADEPT>
- [2] *Basys*, tarjeta con FPGA fabricada por *Digilent*.  
Manual de referencia de la versión E:  
[http://www.digilentinc.com/Data/Products/BASYS/BASYS\\_E\\_RM.pdf](http://www.digilentinc.com/Data/Products/BASYS/BASYS_E_RM.pdf)  
Página web de la tarjeta:  
<http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,791&Prod=BASYS>
- [3] Creative Commons. <http://creativecommons.org/>  
Licencia de este manual: <http://creativecommons.org/licenses/by-nc-nd/3.0/>
- [4] D. Jones, *Stepping motor types*. <http://www.cs.uiowa.edu/~jones/step/index.html>
- [5] Departamento de Tecnología Electrónica, Universidad Rey Juan Carlos. <http://gtebim.es>
- [6] Digilent Inc. <http://www.digilentinc.com/>
- [7] E. Carletti, "Motores paso a paso, características básicas". [http://robots-argentina.com.ar/MotorPP\\_basico.htm](http://robots-argentina.com.ar/MotorPP_basico.htm)
- [8] Electrónica Digital II. <http://gtebim.es/docencia/EDII>
- [9] Ericsson, "Industrial Circuits Application Note: Stepper Motor Basics"  
<http://library.solarbotics.net/pdflib/pdf/motorbas.pdf>
- [10] F. Machado, N. Malpica, J. Vaquero, B. Arredondo, S. Borremeo, "A project-oriented integral curriculum on Electronics for Telecommunication Engineers", EDUCON Conference, Madrid, abril 2010
- [11] F. Machado, S. Borromeo, N. Malpica, "Diseño digital avanzado con VHDL ", Ed. Dykinson, 2009.
- [12] F. Machado, S. Borromeo, N. Malpica, "Diseño digital con esquemáticos y FPGA", Ed. Dykinson, 2009.
- [13] IEEE, *Institute of Electrical and Electronics Engineers*. <http://www.ieee.org>
- [14] IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, IEEE Std 1076.6
- [15] ISE WebPack de Xilinx. <http://www.xilinx.com/tools/webpack.htm>  
Para descargar versiones antiguas:  
<http://www.xilinx.com/webpack/classics/wpclassic/index.htm>
- [16] Minebea. <http://www.minebea.co.jp/english/index.html>
- [17] Mitsumi. [http://www.mitsumi.co.jp/latest/Catalog/indexuse/index\\_e.html](http://www.mitsumi.co.jp/latest/Catalog/indexuse/index_e.html)
- [18] Opencores. <http://www.opencores.org>
- [19] *Pegasus*, tarjeta con FPGA fabricada por *Digilent*. Manual de referencia:  
<http://www.digilentinc.com/Data/Products/PEGASUS/PEGASUS-rm.pdf>  
Página web de la tarjeta:  
<http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,398,537&Prod=PEGASUS>
- [20] Sanken Electric, *Semiconductors General Catalog*. Abril 2010. Disponible:  
[http://www.sanken-ele.co.jp/en/prod/library/lib\\_semicon\\_all.htm](http://www.sanken-ele.co.jp/en/prod/library/lib_semicon_all.htm)  
El apartado de transistores está disponible en:  
[http://www.sanken-ele.co.jp/en/prod/library/images/get\\_pdf.gif](http://www.sanken-ele.co.jp/en/prod/library/images/get_pdf.gif)
- [21] Shinano Kenshi. "Stepper Motor"s. <http://www.shinano.com/xampp/stepper-motors.php>
- [22] Solarbotics. "Stepper Motors".  
[http://www.solarbotics.net/library/pieces/parts\\_mech\\_steppers.html](http://www.solarbotics.net/library/pieces/parts_mech_steppers.html)
- [23] Universidad Rey Juan Carlos, <http://www.urjc.es>
- [24] Wikipedia, Motor paso a paso, [http://es.wikipedia.org/wiki/Motor\\_paso\\_a\\_paso](http://es.wikipedia.org/wiki/Motor_paso_a_paso)
- [25] Xilinx, <http://www.xilinx.com>

- [26] *Xilinx University Program Virtex-II Pro Development System. Hardware reference manual.* UG069 v1.0 9 marzo 2005. <http://www.xilinx.com/univ/xupv2p.html>
- [27] XST User Guide 9.2i. Xilinx, <http://www.xilinx.com/itp/xilinx92/books/docs/xst/xst.pdf>