Diseño de sistemas digitales con VHDL

Felipe Machado Sánchez
Susana Borromeo López
Cristina Rodríguez Sánchez

Departamento de Tecnología Electrónica
Universidad Rey Juan Carlos

Diseño de sistemas digitales con VHDL

Felipe Machado, Susana Borromeo, Cristina Rodríguez

Versión 1.00 creada el 28 de octubre de 2011

Para más información sobre los autores:

http://gtebim.es/

http://gtebim.es/~fmachado

Diseño de sistemas digitales con VHDL

Versión 1.00 creada el 28 de octubre de 2011

Felipe Machado Sánchez
Susana Borromeo López
Cristina Rodríguez Sánchez
Departamento de Tecnología Electrónica
Universidad Rey Juan Carlos
Móstoles, Madrid, España
http://gtebim.es/

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 de la Universidad Rey Juan Carlos por su interés generalizado en aprender y sus comentarios sobre nuestra docencia y prácticas

Los Autores

Índice

Lista de acrónimos

ALU	*Arithmetic Logic Unit* Unidad Aritmético Lógica
BRAM	*Block RAM* Bloque de memoria configurable de las FPGAs de Xilinx
CAD	*Computer Aided Desgin* Diseño asistido por ordenador
CLB	*Configurable Logic Block* Bloque lógico configurable: son los bloques principales de una FPGA donde se implementa un circuito
CPLD	*Complex Programmable Logic Device* Dispositivo de lógica programable complejo
DCSE	*Diseño de Circuitos y Sistemas Electrónicos* Asignatura de la titulación de Ingeniería de Telecomunicación de la URJC
DTE	*Departamento de Tecnología Electrónica*
ED1	*Electrónica Digital I* Asignatura de la titulación de Ingeniería de Telecomunicación de la URJC
ED2	Elec*trónica Digital II* Asignatura de la titulación de Ingeniería de Telecomunicación de la URJC
ETSIT	*Escuela Técnica Superior de Ingeniería de Telecomunicación* Escuela de la Universidad Rey Juan Carlos
FIFO	*First In - First Out* Tipo de pila en la que el primer elemento que ha entrado es el primero en salir
FPGA	*Field Programable Gate Array* Dispositivo de lógica programable, de mayores prestaciones que los CPLD
FSM	*Finite State Machine* Máquina de estados finitos
IEEE	*Institute of Electrical and Electronics Engineers* Instituto de Ingenieros Eléctricos y Electrónicos
ISE	*Integrated Software Environment* Entorno de diseño electrónico de Xilinx. Este libro usa la versión 9.2.
JTAG	*Joint Test Action Group* Nombre común utilizado para designar la norma IEEE 1149.1 titulada *Standard Test Access Port and Boundary-Scan Architecture*
LED	*Light-Emitting Diode* Diodo emisor de luz. Esta palabra se ha lexicalizado en español y se usa en minúsculas. Aunque el plurar correcto sería "ledes", en este libro se empleará leds.
LSB	*Least Significant Bit* Bit menos significativo
LUT	*Look-Up Table*

	Tabla de consulta, en el caso de las FPGA se utilizan para implementar funciones lógicas y se construyen en hardware con multiplexores y elementos de memoria
MSB	*Most Significant Bit*
	Bit más significativo
PISO	*Parallel-In Serial-Out*
	Registro de desplazamiento en el que los datos entran en paralelo y salen en serie
PWM	*Pulse Width Modulation*
	Modulación por ancho de pulso
RAM	*Random-Access Memory*
	Memoria de acceso aleatorio. En general se consideran RAM a las memorias volátiles
RGB	*Red-Green-Blue*
	Rojo, verde y azul. Sistema de codificación del color basado en la mezcla de estos tres colores
ROM	*Read-Only Memory*
	Memoria de sólo lectura.
RTL	*Register-Transfer Level*
	Nivel (de abstracción) de transferencia de registros (o transferencia entre registros)
SIPO	*Serial-In Parallel-Out*
	Registro de desplazamiento en el que los datos entran en serie y salen en paralelo
UART	*Universal Asynchronous Receiver-Transmitter*
	Transmisor-Receptor Asíncrono Universal
URJC	*Universidad Rey Juan Carlos*
	Universidad pública de Madrid, España
UUT	*Unit Under Test*
	Unidad bajo prueba
VHDL	*VHSIC Hardware Description Language*
	Un tipo de lenguaje de descripción de hardware
VHSIC	*Very High Speed Integrated Circuit*
	Circuito integrado de muy alta velocidad

Índice de figuras

Índice de código VHDL

Índice de tablas

1. Introducción

El objetivo de este manual es enseñar a diseñar circuitos y sistemas digitales de cierta complejidad usando VHDL y dispositivos lógicos programables (CPLD o FPGA). El enfoque es aprender a diseñar de manera práctica, y que la necesidad de realizar algo nos haga tener curiosidad por los distintos métodos y técnicas para una eficiente implementación del circuito.

Este manual se ha desarrollado en el Departamento de Tecnología Electrónica [9dte] de la Universidad Rey Juan Carlos para la asignatura Diseño de Circuitos y Sistemas Electrónicos (DCSE [10dcse]) de la titulación de Ingeniería de Telecomunicación[1]. Previamente, los alumnos de esta carrera han cursado las asignaturas Electrónica Digital I (ED1), Electrónica Digital II (ED2 [11ed2]) y Sistemas Electrónicos Digitales (SED), además de otras asignaturas de electrónica básica y electrónica analógica[2].

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 [19mach]. En dichas prácticas se enseña a diseñar circuitos electrónicos digitales con esquemáticos y FPGA. Así que para seguir este manual suponemos que ya has adquirido los 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 [19mach] es una buena base para esto.

Un año más tarde, en ED2 los alumnos aprendieron a diseñar circuitos digitales más complejos, teniendo que dominar el empleo de máquinas de estados finitos (FSM[3]) para el diseño y análisis de los circuitos digitales. En la práctica, se pasó de diseñar con esquemáticos a diseñar con el lenguaje de descripción de hardware VHDL. Los diseños de los circuitos digitales de este curso están recogidos en el manual de la asignatura [17mach]. Ejemplos de los diseños de dicho manual son: cronómetros, el control de máquinas expendedoras, control de motores, claves electrónicas, etc.

Por tanto, para seguir este manual con una mayor comprensión se recomienda haber realizado los diseños propuestos en manual de ED2 [17mach]. Este manual está disponible en el archivo abierto de la Universidad Rey Juan Carlos. De todos modos, si sólo necesitas un breve repaso de VHDL, el manual que estás leyendo incluye una pequeña introducción a este lenguaje de descripción de hardware.

Como seguramente recuerdes, el VHDL es un estándar del Instituto de Ingenieros Eléctricos y Electrónicos[4] (IEEE [13ieee]). Existen otros lenguajes de descripción de hardware como el Verilog o el SystemC. Históricamente, la existencia simultánea del VHDL y Verilog ha dividido a la comunidad de diseñadores, lo que ha provocado

[1] 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

[2] En la referencia [12educ] se resume el currículum en electrónica que adquieren los alumnos en dicho plan de estudios

[3] FSM: acrónimo del inglés: *Finite State Machine*: Máquina de estados finitos

[4] *Institute of Electrical and Electronics Engineers*

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** [25rtl]. 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, lo que hace ahorrar mucho tiempo a los diseñadores de circuitos. De la misma manera que en el manual de ED2 [17mach], 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 propondremos sólo una de ellas.

Para realizar las prácticas propuestas en este manual se utilizarán dos tarjetas basadas en FPGA de Xilinx [29xilinx]: la *XUP Virtex II pro* [31xup] y la *Nexys2* [21nexys]. Actualmente Xilinx es el mayor fabricante de FPGA seguido de Altera. Xilinx proporciona una herramienta de diseño asistido por ordenador (CAD) gratuita: el *ISE Webpack* [16ise] que será la que se use en este manual. Para la simulación, se utilizará el propio simulador del ISE o el Modelsim [20model]. Sin embargo, aunque con un esfuerzo mayor, podrás seguir este manual usando herramientas y tarjetas diferentes si te abstraes de las particularidades de éstas.

Al realizar este manual hemos intentado buscar un punto intermedio entre dar una solución a cada práctica planteada, y dejarte sólo y sin ayuda para que encuentres tu propia solución. Así que hemos intentado dar un número de pautas suficientes para que puedas realizar el diseño por ti mismo sin que te sobrepase la dificultad, o que por el contrario, copies y pegues la solución propuesta sin que te suponga un reto o que no te estés dando cuenta de lo que estás haciendo. Encontrar ese punto intermedio no es fácil, y puede ser que a veces te parezca que el grado de detalle es excesivo, o que por el contrario, necesites más ayuda. En ambos casos, te pedimos que entiendas la dificultad que nos supone encontrar ese punto medio, y que además no es el mismo para todos los que van a leer este manual.

Las primeras prácticas son más detalladas y conforme se avanza se proporciona menos información de detalle y se incide en los aspectos más generales de cómo abordar el diseño. Las prácticas son de dificultad incremental y por este motivo te recomendamos que sigas el orden establecido. A veces se incluyen apartados opcionales, que puedes saltártelos si así lo deseas. Estos apartados se han señalado con el símbolo ∗.

Es muy importante que para seguir este manual realices las prácticas y las implementes, y no te limites a leerlas. Ya sabes que la práctica no siempre es igual que la teoría. También te recomendamos que intentes pensar un tu propia solución antes de mirar las guías ofrecidas para abordar el diseño. Pensar en una solución y compararla con la que aquí se propone te ayudará a aprender a diseñar.

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 una licencia *Creative Commons* [5cc] que permite su copia y distribución. Hemos puesto el

libro en el archivo abierto de la Universidad Rey Juan Carlos y se puede acceder a él desde el siguiente enlace http://hdl.handle.net/10115/5700, en donde además de este libro puedes descargar algunos ficheros VHDL de las prácticas de este libro. Estos ficheros también están disponibles en el enlace [28web]

Esperamos que disfrutes de su lectura y que te ayude a aprender a diseñar circuitos y sistemas 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, susana.borromeo@urjc.es y cristina.rodriguez.sanchez@urjc.es

2. Repaso de VHDL para síntesis

En este capítulo repasaremos descripciones simples de VHDL sintetizable. Aunque suponemos que se conocen los conceptos aquí explicados, este capítulo se podrá usar como referencia para algunas de las construcciones VHDL más típicas. Si tienes dudas o quieres repasar con más profundidad lo que aquí se dice te recomendamos un manual más básico [17mach].

El VHDL es un lenguaje muy amplio y fue concebido inicialmente para modelado y simulación, no para síntesis. Por tanto, no todas las descripciones VHDL son sintetizables, esto es, no todas las descripciones tienen una equivalencia en el nivel de puertas. Por otro lado, una misma funcionalidad puede describirse de muchas maneras.

Es por esto que este capítulo no pretende ser un manual exhaustivo del VHDL, sino dar *recetas* para facilitar el diseño en VHDL. Todos los ejemplos de este manual se pueden describir de otra manera.

Antes de empezar con los ejemplos recordemos que el VHDL no distingue entre mayúsculas y minúsculas. De todos modos, por claridad se recomienda mantener una misma forma de escribir. Esto es, si por ejemplo se ponen los operadores en mayúsculas, sería recomendable que se mantuviesen así en todo el diseño, lo mismo es válido para los nombres de señales.

Hay un conjunto de palabras reservadas que no se deben usar para otros propósitos (por ejemplo para nombres de señales). Éstas son:

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 2.1: Lista de palabras reservadas del VHDL

A continuación veremos una serie de ejemplos de diseños VHDL de dificultad incremental.

2.1. Puerta AND

Para diseñar un circuito en VHDL lo mínimo que se necesita es una **entidad** y una **arquitectura**. La entidad define el circuito como una caja, con sus puertos de entrada y salida. Mientras que la arquitectura describe el interior del circuito. Una misma entidad puede tener varias arquitecturas.

Así, si queremos describir una puerta `AND` en VHDL, crearemos la entidad:

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

entity PUERTA_AND is
  port (
    A : in   std_logic;
    B : in   std_logic;
    C : out  std_logic
  );
end;
```

Código 2.1: Entidad en VHDL

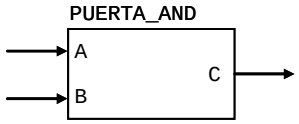


Figura 2.2: Representación de la entidad VHDL

Hay varias cosas que se deben resaltar:

- La entidad no se podría llamar `AND` ya que es una palabra reservada (ver figura 2.1).
- Aunque la entidad se llame `PUERTA_AND`, simplemente representa su nombre, pero no su funcionalidad. Que tenga ese nombre no significa que vaya a tener que funcionar como una puerta `AND`. Dependerá de lo que se describa en la arquitectura. Sin embargo, como es lógico, no es recomendable ponerle un nombre que no describa su funcionalidad.
- Los puertos de entrada se han declarado de tipo `std_logic`. Este es un tipo de datos de un solo bit que puede tener valores distintos de '0' y '1'. Y se usan así para tener más información en la simulación. Los valores que puede tener un `std_logic` son:

`'U'` : No ha recibido valor	`'X'` : Ha recibido valores distintos (desconocido *fuerte*)	`'0'` : Cero lógico (cero *fuerte*)
`'1'` : Uno lógico (uno fuerte)	`'Z'` : Alta impedancia	`'W'` : Desconocido débil
`'L'` : Cero débil	`'H'` : Uno débil	`'-'` : No importa

Tabla 2.1: Valores del `std_logic`

Alternativamente se podían haber declarado como tipo `bit`. El tipo `bit` sólo tiene los valores '0' y '1'. Sin embargo es más recomendable utilizar el tipo `std_logic`, pues en simulación, es importante saber si la señal no ha recibido valor (`'U'`), si está recibiendo valores distintos en un mismo tiempo (`'X'`) o si toma el valor de alta impedancia (`'Z'`) que veremos en el apartado 11.2.4.

Para poder usar el tipo `std_logic` hay que indicar que se usa el paquete `std_logic_1164`, y esto se hace con las dos líneas que están antes de la entidad (ver código 2.1).

Como hemos dicho, por ahora nuestra entidad no hace nada, simplemente hemos definido sus entradas y salidas. Para describir su funcionalidad usamos la arquitectura:

```
architecture BEHAVIORAL of PUERTA_AND
is
begin
  C <= A and B;
end BEHAVIORAL;
```

Código 2.2: Arquitectura en VHDL

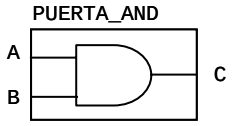


Figura 2.3: Representación de la arquitectura VHDL

El nombre de la arquitectura es `BEHAVIORAL` (comportamental). Normalmente se pone un nombre que describa el nivel de abstracción de la descripción, que puede ser `dataflow`, `functional`, `RTL`,... Pero no tiene por qué ser uno de estos.

Del código podemos deducir que la asignación de señales se realiza mediante el operador: "`<=`".

2.2. Varias puertas

Las puertas disponibles en VHDL son: `AND`, `NAND`, `OR`, `NOR`, `XOR`, `XNOR`, `NOT`.

Para implementar un circuito con varias puertas, éstas se pueden poner en la misma expresión. Por ejemplo, en el código 2.3 se muestra una arquitectura que tiene una única sentencia con todas las puertas. Para simplificar, en el código no se ha incluido la entidad (y así se hará en los ejemplos sucesivos a no ser que sea necesario).

```
architecture UNO of PUERTAS is
begin
  Z <= (not(A or B)) xor C;
end UNO;
```

Código 2.3: Arquitectura en VHDL

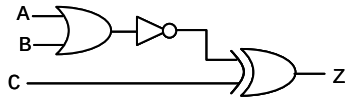


Figura 2.4: Representación de la arquitectura VHDL

Existen reglas de precedencia entre los operadores, pero lo más seguro es poner paréntesis, puesto que además de facilitar la lectura, nos aseguramos que lo estamos describiendo correctamente.

Se puede realizar un circuito equivalente al código 2.3 en dos sentencias, para ello es necesario usar señales auxiliares:

```
architecture DOS of PUERTAS is
  signal  AUX : std_logic;
begin
  AUX <= not(A or B);
  Z <= AUX xor C;
end DOS;
```

Código 2.4: Arquitectura en VHDL

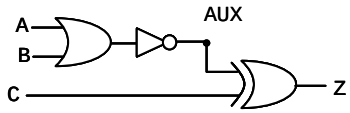


Figura 2.5: Representación de la arquitectura VHDL

Un concepto muy importante en hardware y por tanto en VHDL es la **concurrencia**. Esto implica que **no importa el orden de las sentencias** que están en el cuerpo de una arquitectura, ya que el hardware se ejecuta concurrentemente: El hardware siempre se está ejecutando. Así que el código 2.4 es equivalente al siguiente código:

```
architecture TRES of PUERTAS is
  signal  AUX : std_logic;
begin
  Z <= AUX xor C;       -- Ahora esta sentencia se ha puesto antes que la siguiente
  AUX <= not(A or B);   -- en VHDL el orden de sentencias concurrentes es indiferente
end TRES;
```

Código 2.5: Arquitectura equivalente a la del código 2.4

Antes de seguir, asegúrate que entiendes por qué los códigos 2.4 y 2.5 son equivalentes.

2.3. Multiplexor

El multiplexor selecciona entre varias alternativas según el valor de la señal de selección. Para ello el VHDL ofrece varias construcciones posibles.

Se puede realizar mediante una sentencia concurrente en la que se incluya condición:

```
architecture UNO of MUX is

begin

  Z <= A when Sel='1' else B;

end UNO;
```

Código 2.6: Arquitectura de un multiplexor en VHDL

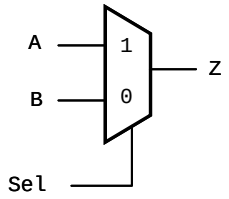


Figura 2.6: Representación de la arquitectura VHDL

Otra manera de realizar un multiplexor es mediante el uso de procesos. Dentro de un proceso la ejecución de las sentencias es **secuencial**:

```
architecture DOS of MUX is
begin

  P_MUX: process (A,B,Sel)        -- (A,B,Sel) es la lista de sensibilidad
  begin
    if Sel='1' then
      Z <= A;
    else
      Z <= B;
    end if;
  end process;

end DOS;
```

Código 2.7: Arquitectura equivalente de un multiplexor en VHDL

En el código 2.7 se muestra un multiplexor equivalente. Para ello se ha usado un proceso, que como se ha dicho, se ejecutan de forma secuencial. Los procesos tienen una lista de sensibilidad, en la que se tienen que incluir todas las señales leídas dentro del proceso. En este caso, las señales leídas son `A`, `B` y `Sel`.

Hemos realizado el multiplexor con la sentencia `if`, aunque es más habitual hacerla con la sentencia `case`. En este caso no es muy importante porque la señal de selección (`Sel`) es de un solo bit.

```
architecture DOS of MUX is
begin
  P_MUX: process (A,B,Sel)
  begin
    case Sel is       -- en el case tienen que estar presentes todas las alternativas
      when '1' =>
        Z <= A;
      when others =>  -- si no pongo todas las alternativas, pongo "others"
        Z <= B;
    end case;
  end process;
end DOS;
```

Código 2.8: Descripción del multiplexor equivalente a la del código 2.7 utilizando la sentencia case

En el código 2.8 se ha descrito un multiplexor con una sentencia `case`. La sentencia `case` tiene que tener todas las alternativas, pudiendo haber una alternativa por defecto llamada `"others"`. Esta alternativa engloba a todas las que faltan. En este caso, como estamos usando datos de tipo `std_logic`, tenemos que usar el `others`, ya que hay más alternativas que '0' y '1' (ver tabla 2.1).

2.4. Multiplexor de 2 bits de selección

El VHDL permite usar tipos de datos de más de un bit (vectores). Para los vectores se puede emplear el tipo `std_logic_vector`. Los vectores se declaran desde el bit más significativo (de mayor índice) hasta el menos significativo (al que se le da el índice cero[5]).

En el código 2.9 se muestra la descripción de un multiplexor de 2 bits de selección (con datos de un bit). Fíjate en la declaración de la señal de selección: `Sel`. Para esta señal se usa un `std_logic_vector` cuyo rango va de 1 a 0. El bit 1 es el más significativo.

```
entity MUX_2B_SEL is
  port (
    A     : in   std_logic;
    B     : in   std_logic;
    C     : in   std_logic;
    Sel   : in   std_logic_vector (1 downto 0);
    Z     : out  std_logic
  );
end;

architecture UNO of MUX_2B_SEL is
begin
  P_MUX: process (A,B,C,SEL)
  begin
    case Sel is
      when "00" =>
        Z <= A;
      when "01" | "10" =>   -- mismo valor para
        Z <= B;             -- las 2 opciones
      when others =>
        Z <= C;
    end case;
  end process;
end UNO;
```

Código 2.9: Multiplexor de 2 bits de selección

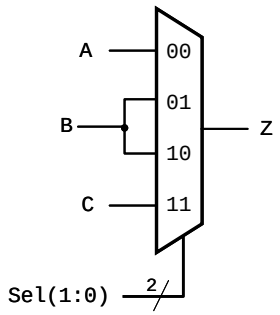


Figura 2.7: Representación del multiplexor

También es interesante ver en el código 2.9 cómo se indica en una sentencia `case` que dos opciones dan el mismo resultado. Para ello se utiliza el operador "`|`".

El proceso del mismo multiplexor descrito con una sentencia `if` en vez de case quedará como se muestra en el código 2.10. Este código produce un multiplexor equivalente al del código 2.9.

```
P_MUX_2: process (A,B,C,Sel)
begin
  if Sel = "00" then
    Z <= A;
  elsif (Sel = "01") OR (Sel = "10") then
    Z <= B;
  else
    Z <= C;
  end if;
end process;
```

Código 2.10: Proceso equivalente para el multiplexor de 2 bits de selección del código 2.9

2.5. Elementos de memoria: latch[6]

Para describir un elemento de memoria podemos utilizar un proceso con sentencias condicionales en el que en algunas alternativas dejemos la señal sin asignar. Debido a que en dichas condiciones no se asigna ningún valor a la señal, ésta tendrá que guardar el

[5] Aunque se puede poner un índice distinto de cero, no es recomendable.

[6] Un *latch* es un elemento de memoria con dos posibles estados (biestable) que es activo por nivel. La señal que hace que el *latch* esté activo se suele denominar *enable* (habilitación).

valor que anteriormente tenía. Para guardar el valor será necesario utilizar un elemento de memoria.

En el código 2.11 se muestra la descripción de un *latch* de tipo D activo por nivel alto. Cuando la señal `Enable` vale '0', la señal `Z` no recibe valor, y por lo tanto tendrá que guardar el dato que tenía antes.

```
entity LATCH is
  port (
    Enable  : in   std_logic;
    A       : in   std_logic;
    Z       : out  std_logic
  );
end;

architecture BEHAVIORAL of LATCH is
begin
  P_LATCH: process (Enable, A)
  begin
    if Enable = '1' then
      Z <= A;
    end if;
  end process;
end BEHAVIORAL;
```

Código 2.11: Latch *de tipo D activo por nivel alto*

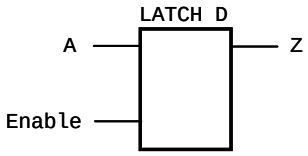


Figura 2.8: Latch *de tipo D*

Es muy importante tener esto presente, ya que cuando hay condiciones en las que no se le asigna valor a una señal se genera un *latch*. A veces éstos se generan por error del diseñador, que no se ha fijado que en alguna alternativa no ha asignado ningún valor a la señal.

El código 2.11 se puede modificar para incluir una señal de *reset* asíncrono (código 2.12).

```
entity LATCH2 is
  port (
    ResetN  : in   std_logic;
    Enable  : in   std_logic;
    A       : in   std_logic;
    Z       : out  std_logic
  );
end;
architecture BEHAVIORAL of LATCH2 is
begin
  LATCH: process (ResetN, Enable, A)
  begin
    if ResetN = '0' then    -- activo por nivel bajo
      Z <= '0';
    elsif Enable = '1' then
      Z <= A;
    end if;
  end process;
end BEHAVIORAL;
```

Código 2.12: Latch *de tipo D activo por nivel alto con reset*

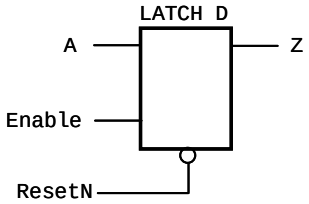


Figura 2.9: Latch *de tipo D con reset*

Fíjate que el reset se ha llamado `ResetN`. Con la "N" se ha querido indicar que es activo por nivel bajo, esto es, que cuando el reset vale cero, se resetea el *latch*.

2.6. Elementos de memoria: biestables activos por flanco

Para describir un elemento de memoria activo por flanco se utilizan las siguientes expresiones:

- Para biestable activo por flanco de subida: `clk'event and clk = '1'`
- Para biestable activo por flanco de bajada: `clk'event and clk = '0'`

En el código 2.13 se muestra la descripción de un biestable D activo por flanco de subida. En la lista de sensibilidad de los biestables se puede poner solamente las señales que se asignan (o sean condición) antes de la sentencia de reloj. No es un error poner todas las señales que influyen, sin embargo puede ser más cómodo evitarnos poner todas, además de que hacemos que la simulación sea más rápida.

```
entity BIESTABLE is
  port (
    ResetN  : in   std_logic;
    Clk     : in   std_logic;
    A       : in   std_logic;
    Z       : out  std_logic
  );
end;
architecture BEHAVIORAL of BIESTABLE is
begin
  P_BIEST:process (ResetN, Clk) --solo reset y clk
  begin
    if ResetN = '0' then
      Z <= '0';
    elsif Clk'event and Clk = '1' then
      Z <= A;
    end if;
  end process;
end BEHAVIORAL;
```

Código 2.13: Biestable de tipo D activo por flanco de subida

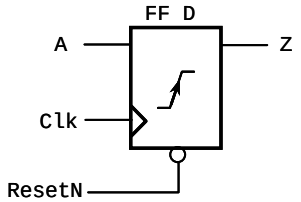


Figura 2.10: Biestable de tipo D con reset

2.7. Actualización de los valores de las señales

En un proceso, el valor de la señal se actualiza al finalizar el proceso y no durante la secuencia de asignaciones. En el código 2.14, en cada flanco de subida del reloj, a la señal `B` se le asigna `A`, y a `Z` se le asigna `B`. Sin embargo, `Z` no recibe el valor de `A` directamente, sino que recibe el valor que tenía `B` cuando se entró en el proceso y no el que va a recibir. Esto es, a `Z` le llega el valor de `A` con un retardo de un ciclo de reloj.

```
entity DETECTA_FLANCO is
  port (
    ResetN  : in   std_logic;
    Clk     : in   std_logic;
    A       : in   std_logic;
    Z       : out  std_logic
  );
end;
architecture UNO of DETECTA_FLANCO is
  signal B : std_logic;
begin
  P_DETECTA: process (ResetN, Clk)
  begin
    if ResetN = '0' then
      B <= '0';
      Z <= '0';
    elsif Clk'event and Clk = '1' then
      B <= A;
      Z <= B;  -- B no toma el valor de A
               -- inmediatamente
    end if;
  end process;
end UNO;
```

Código 2.14: Dos biestables en cascada

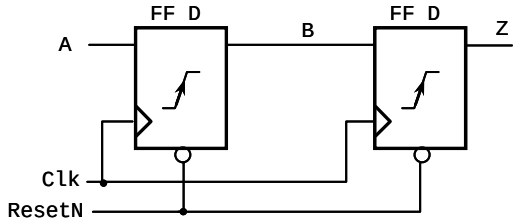


Figura 2.11: Dos biestables en cascada

En la figura 2.11 se muestra el circuito generado por el código 2.14. Este código es equivalente asignar las señales en diferentes procesos. Esto se muestra en el código 2.15 (sólo aparece la arquitectura).

```
architecture DOS of DETECTA_FLANCO is
  signal B : std_logic;
begin

  P_BIEST_1: process (ResetN, Clk)
  begin
    if ResetN = '0' then
      B <= '0';
    elsif Clk'event and Clk = '1' then
      B <= A;
    end if;
  end process;

  P_BIEST_2: process (ResetN, Clk)
  begin
    if ResetN = '0' then
      Z <= '0';
    elsif Clk'event and Clk = '1' then
      Z <= B;
    end if;
  end process;

end DOS;
```

Código 2.15: Diseño equivalente al código 2.14

A continuación se muestra cómo con unas pequeñas modificaciones del circuito anterior se puede crear un circuito detector de flanco de subida[7].

```
entity DETECTA_FLANCO is
  port (
    ResetN    : in   std_logic;
    Clk       : in   std_logic;
    A         : in   std_logic;
    FlancoUp  : out  std_logic
  );
end;

architecture TRES of DETECTA_FLANCO is
  -- ahora Z no es de salida
  signal B, Z : std_logic;
begin

  P_BIEST_1: process (ResetN, Clk)
  begin
    if ResetN = '0' then
      B <= '0';
    elsif Clk'event and Clk = '1' then
      B <= A;
    end if;
  end process;

  P_BIEST_2: process (ResetN, Clk)
  begin
    if ResetN = '0' then
      Z <= '0';
    elsif Clk'event and Clk = '1' then
      Z <= B;
    end if;
  end process;

  FlancoUp <= '1' when Z='1' and B='0' else
              '0';
end TRES;
```

Código 2.16: Circuito detector de flanco

[7] Los circuitos detectores de flanco se explican en la referencia [17mach]

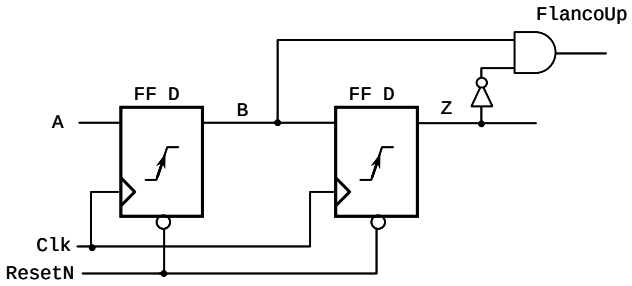


Figura 2.12: Esquema del diseño del código 2.16

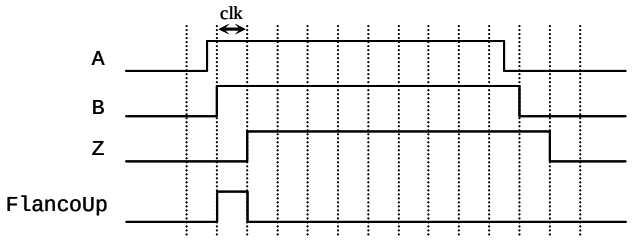


Figura 2.13: Cronograma del diseño del código 2.16

2.8. Contadores

Para contar podemos utilizar tipos de datos enteros (`integer`) o naturales (`natural`). Es muy importante definir el rango del número, porque si no se sintetizará un número de 32 bits, lo que implica 32 cables. Es también importante hacer que el rango sea un número potencia de 2. Para que coincida el rango que definimos con el que se sintetizará. Por ejemplo, si creamos un rango de 0 a 13, el sintetizador creará un número de 4 bits (de 0 a 15), y por lo tanto, nuestro número podrá tomar valores que quedarán fuera del rango que hemos especificado (14 y 15).

Un contador de módulo 8 se muestra en el código 2.17

```
entity CONTADOR is
  port (
    rst_n   : in    std_logic;
    clk     : in    std_logic;
    cuenta  : out natural range 0 to 7
  );
end CONTADOR;

architecture BEHAVIORAL of CONTADOR is
  signal cuentaaux : natural range 0 to 7;  -- rango potencia de 2: 8 (de 0 a 7)
begin
  cuenta <= cuentaaux;                    -- Se conecta cuentaaux a cuenta
  P_CONT: Process (rst_n, clk)
  begin
    if rst_n = '0' then
      cuentaaux <= 0;
    elsif clk'event and clk='1' then
      if cuentaaux = 7 then      -- cuenta no se podria leer por ser puerto de salida
        cuentaaux <= 0;          -- por eso se usa cuentaaux
      else
        cuentaaux <= cuentaaux + 1;      -- Aqui se lee cuentaaux otra vez
      end if;
    end if;
  end process;
end BEHAVIORAL;
```

Código 2.17: Contador módulo 8

En el código 2.17 hay un nuevo concepto que se debe tener en cuenta al diseñar en VHDL: **Los puertos de salida no se pueden leer**. Por esto se ha tenido que crear la señal auxiliar

cuentaaux. Con cuentaaux se realizan las operaciones, y en una sentencia concurrente se conecta con la señal cuenta.

Para más una explicación más detallada de los contadores se recomienda la referencia [17mach].

2.9. Registros de desplazamiento

Un registro de desplazamiento se puede describir como se muestra en el código 2.18. Este circuito carga en paralelo el dato datoin cuando carga='1'. Y desplaza el dato a través del bit 0 en serie.

Una vez más se recomienda la referencia [17mach] para profundizar en los registros de desplazamiento.

```
entity REG_DESPLZ is
  port (
    rst_n     : in    std_logic;
    clk       : in    std_logic;
    carga     : in    std_logic;
    desplz    : in    std_logic;
    datoin    : in    std_logic_vector (7 downto 0);
    serieout  : out   std_logic
  );
end REG_DESPLZ;

architecture BEHAVIORAL of REG_DESPLZ is
  signal datoreg : std_logic_vector (7 downto 0);
begin
  serieout <= datoreg(0);

  REG_DESPLZ:Process (rst_n, clk)
  begin
    if rst_n='0' then
      datoreg <= (others=>'0');
    elsif clk'event and clk='1' then
      if carga = '1' then
        datoreg <= datoin;               -- Carga paralelo
      elsif desplz ='1' then             -- Desplazamiento
        datoreg(6 downto 0) <= datoreg(7 downto 1);
      end if;
    end if;
  end process;
end architecture;
```

Código 2.18: Registro de carga paralelo y salida serie

2.10. VHDL estructural

Para diseñar un circuito grande es conveniente separarlo en bloques funcionales. Cada uno de estos bloques se diseña como hemos visto hasta ahora (entidad y arquitectura). Y estos bloques se juntan en una arquitectura que hace referencia a estos bloques. Habitualmente este tipo de arquitectura se denomina Estructural. En un diseño grande puede haber varios niveles de jerarquía, donde unas arquitecturas estructurales contienen a otras. Las ventajas de realizar el diseño de esta manera son varias:

- Permite dividir el circuito en partes más sencillas, haciendo que el diseño sea más manejable y entendible. Esto además facilita el trabajo en equipo.
- Es fácil modificar algún bloque del diseño, mientras que se deja el resto igual.
- Puede pasar que en un diseño se necesite usar varios bloques iguales, realizando el diseño estructural, no hace falta volver a repetir el bloque, sino que basta con describir varias referencias a ese bloque.

- Permite la reutilización ya que puede que un bloque de un diseño lo necesitemos para otro diseño.
- Podemos ir probando cada uno de los bloques al irlos diseñando y una vez comprobado que funcionan bien, irlos uniendo.

Para explicar cómo se describe una arquitectura estructural en VHDL lo haremos con un ejemplo. Supongamos que vamos a realizar un circuito con los siguientes componentes:

- El multiplexor de la figura 2.6 (el código de la arquitectura está en el código 2.7)
- Dos biestables como el de la figura 2.10 (código 2.13)
- El registro de desplazamiento del código 2.18.

El esquema del circuito estructural que queremos realizar se muestra en la figura 2.14. En éste se han dibujado los bloques internos del circuito como cajas. Cada bloque tiene un nombre `REFERENCIA:ENTIDAD`. El primer nombre (`REFERENCIA`) indica el nombre único que identifica a ese bloque en particular. El segundo nombre indica el componente al que se refiere (el nombre de la entidad). Puede que un componente se utilice más de una vez, entonces este nombre estará repetido (como es el caso del `BIESTABLE`). Por tanto, la referencia será única, aunque un mismo componente se utilice más de una vez. La referencia a veces se llama instancia o *instanciación*, ya que en inglés se llama *component instantiation*.

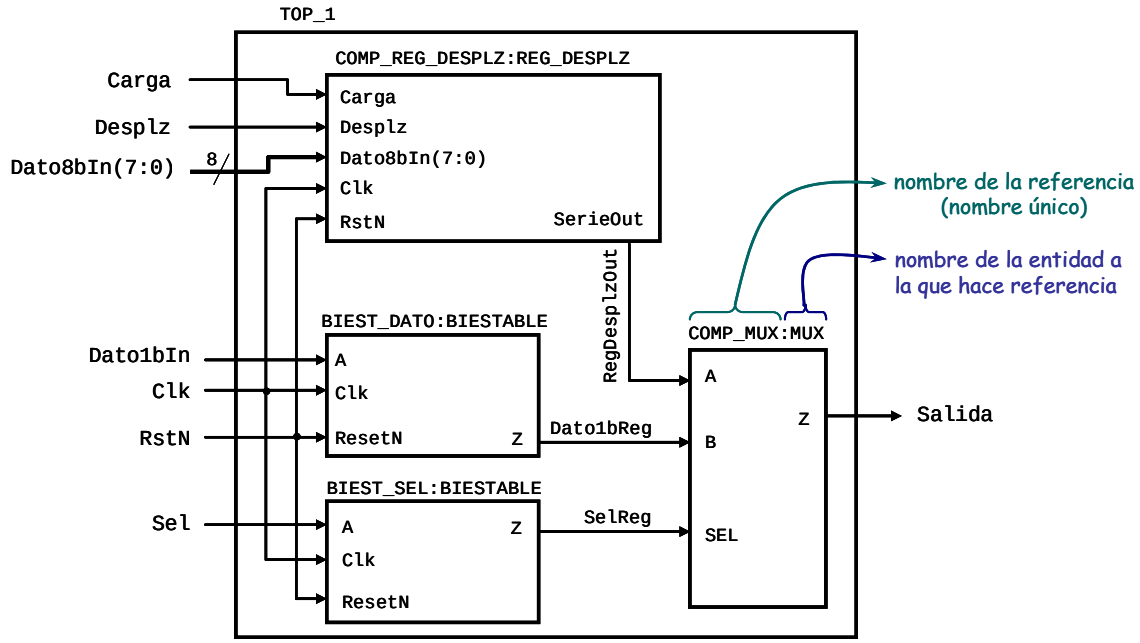


Figura 2.14: Esquema del circuito estructural

El código VHDL del circuito de la figura 2.14 se muestra en el código 2.19. Dentro de la parte declarativa de la arquitectura (antes de su `begin`) se declaran los componentes que se van a utilizar. Esta declaración de los componentes es similar a la descripción de la entidad del componente. Fíjate que se declaran tres componentes en vez de cuatro, ya que hay uno (`BIESTABLE`) que se usa dos veces. En esta parte declarativa se declaran también las señales internas que harán las conexiones (`SelReg`, `Dato1bReg`, `RegDezplzOut`).

En la parte de sentencia de la arquitectura (después del `begin`) se ponen las referencias de los componentes. Estas referencias especifican las conexiones de los puertos del componente con las señales y puertos de la arquitectura. En la referencia al componente,

su nombre debe ser único: `COMP_REG_DESPLZ`, `BIEST_DATO`, `BIEST_SEL`, `COMP_MUX`. Y a continuación, después de dos puntos, va el nombre del componente (entidad). En caso de que se utilice un mismo componente dos veces (como ocurre con `BIESTABLE`) este nombre estará repetido.

En VHDL se puede utilizar un componente con un nombre distinto a la entidad a la que se refiere. Para ello hay sentencias que lo especifican. Sin embargo, no todos los sintetizadores contemplan esta posibilidad, y lo más recomendable es declarar los componentes (y sus puertos) con el mismo nombre que su entidad.

El VHDL también permite especificar qué arquitectura elegir para el componente (si hubiese más de una), sin embargo, muchas herramientas cogen automáticamente la última arquitectura compilada independientemente de si se especifica otra arquitectura.

```
entity TOP_1 is
  port (
    RstN     : in    std_logic;
    Clk      : in    std_logic;
    Carga    : in    std_logic;
    Desplz   : in    std_logic;
    Sel      : in    std_logic;
    Dato1bIn : in    std_logic;
    Dato8bIn : in    std_logic_vector (7 downto 0);
    Salida   : out   std_logic
  );
end TOP_1;

architecture ESTRUCTURAL of TOP_1 is

  -- Declaracion del componente registro de deplazamiento
  component REG_DESPLZ
  port (
    RstN     : in    std_logic;
    Clk      : in    std_logic;
    Carga    : in    std_logic;
    Desplz   : in    std_logic;
    DatoIn   : in    std_logic_vector (7 downto 0);
    SerieOut : out   std_logic
  );
  end component;

  -- Declaracion del componente multiplexor
  component MUX
  port (
    A     : in   std_logic;
    B     : in   std_logic;
    Sel   : in   std_logic;
    Z     : out  std_logic
  );
  end component;

  -- Declaracion del componente biestable
  component BIESTABLE
  port (
    ResetN  : in   std_logic;
    Clk     : in   std_logic;
    A       : in   std_logic;
    Z       : out  std_logic
  );
  end component;

  -- Declaracion de las senales que realizan las interconexiones
  signal RegDesplzOut   : std_logic;
  signal Dato1bReg      : std_logic;
  signal SelReg         : std_logic;
```

```
begin

  -- Referencia al componente RG_DESPLZ, la referencia se llama COMP_REG_DESPLZ
  COMP_REG_DESPLZ: REG_DESPLZ
    port map (
      --(Puerto interno del componente) => (Senal o puerto de arquitectura)
      RstN      => RstN,
      Clk       => Clk,
      Carga     => Carga,
      Desplz    => Desplz,
      DatoIn    => Dato8bIn,
      SerieOut  => RegDesplzOut
    );

  -- Referencia al componente Biestable
  BIEST_DATO: BIESTABLE
    port map (
      ResetN    => RstN,
      Clk       => Clk,
      A         => Dato1bIn,
      Z         => Dato1bReg
    );

  -- Segunda referencia al componente Biestable
  BIEST_SEL: BIESTABLE
    port map (
      ResetN    => RstN,
      Clk       => Clk,
      A         => Sel,
      Z         => SelReg
    );

  COMP_MUX: MUX
    port map (
      A         => RegDesplzOut,
      B         => Dato1bReg,
      Sel       => SelReg,
      Z         => Salida
    );
end architecture;
```

Código 2.19: Ejemplo de diseño estructural

En el apartado 7.6.3 veremos otro ejemplo de diseño estructural.

3. Funcionamiento básico de las tarjetas electrónicas

El objetivo de esta práctica guiada es introducir y empezar a usar la placa *XUP Virtex-II Pro* (XUPV2P) [31xup] y la placa *Nexys 2* [21nexys] mediante el uso de funcionalidad elemental ya utilizada en otras asignaturas, como son los pulsadores, interruptores y LED.

En el apartado siguiente veremos las características generales de estas dos tarjetas. Posteriormente implementaremos un diseño sencillo para asegurarnos de que llevamos a cabo todo el proceso de diseño con normalidad. Este paso es importante antes de seguir con diseños más complejos.

3.1. Tarjetas XUPV2P y Nexys2

La tarjeta utilizada en clase será la *XUPV2P* de la empresa *Digilent* [8digi]. A fecha de la redacción de este manual, esta tarjeta ha dejado de producirse aunque se sigue vendiendo. Esta tarjeta tiene un importante descuento importante si se va a utilizar con fines académicos.

También explicaremos cómo utilizar la tarjeta *Nexys2*, que es más sencilla, tiene un precio más asequible y sigue en producción.

Cuando tengas las tarjetas, debes evitar tocar los componentes metálicos y conexiones para no dañar los circuitos por la electricidad estática de nuestro cuerpo, ni siquiera estando desconectada de la fuente.

3.1.1. La tarjeta XUPV2P

Antes de iniciar la práctica se recomienda observar la placa y se identificar físicamente cada una de los componentes señalados en la figura 3.1.

La tarjeta XUPV2P tiene una FPGA de Xilinx de modelo *Virtex-II Pro 30*. Esta FPGA tiene 30816 celdas lógicas[8] y más de 500 pines disponibles para el usuario, aparte de bloques de memoria, multiplicadores y otros bloques.

A medida que vayamos avanzando, iremos viendo los componentes de la placa, así como algunos de los bloques internos de la FPGA.

[8] Una celda lógica consiste en una LUT (*Look-Up Table*) de 4 entradas, un biestable y lógica de acarreo

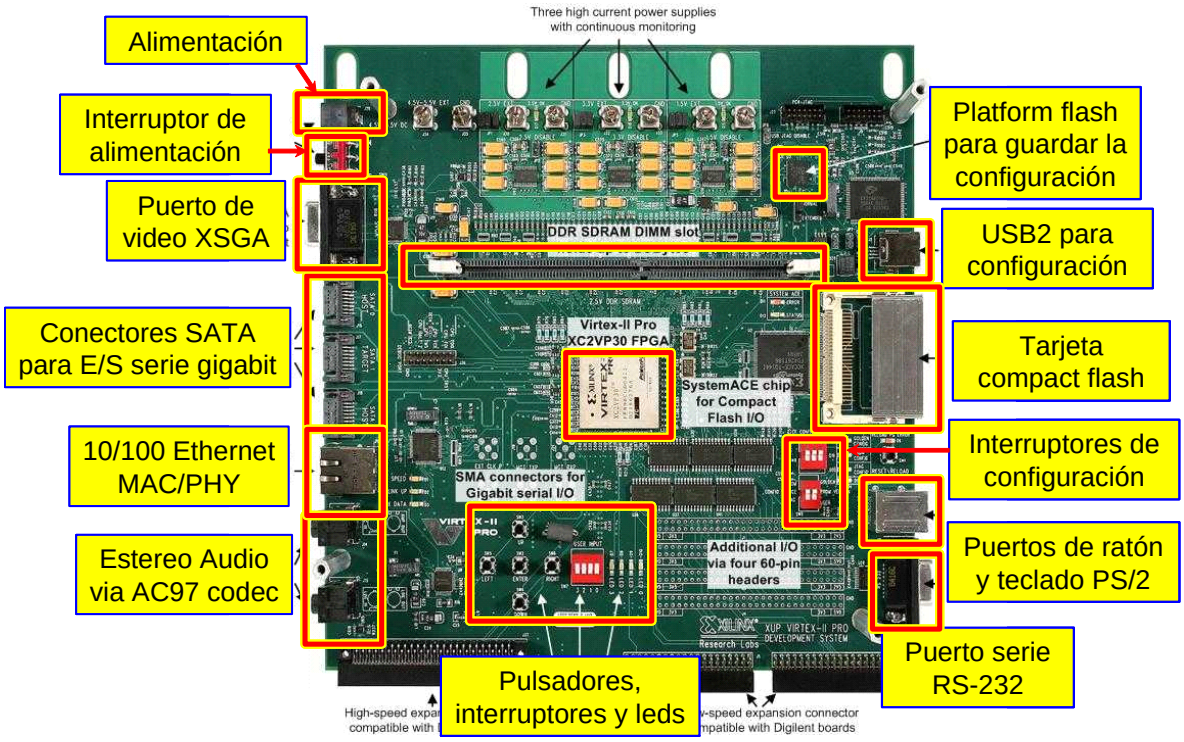


Figura 3.1: Placa XUP Virtex-II Pro (XUPV2P) utilizada en la asignatura

En las prácticas de este capítulo se utilizarán los pulsadores, interruptores y LED que la XUPV2P dispone para uso general. La placa además dispone de otros pulsadores, interruptores y LED que tienen una funcionalidad específica. En la parte baja de la figura 3.1, bajo el epígrafe "*Pulsadores, interruptores y leds*" puedes ver dónde están los que vamos a utilizar.

En la figura 3.2 se muestran las conexiones en la placa de los pulsadores, interruptores, y LED. Analizando la figura podemos observar que al pulsar o conectar un interruptor enviamos un '0' al pin de la FPGA, y de manera análoga, los LED se encienden al poner un '0' en el pin de la FPGA correspondiente. Por lo tanto, en la **tarjeta XUPV2P se trabaja con lógica negada con los pulsadores, interruptores y LED**. Veremos que esto es muy importante tenerlo en cuenta, sobre todo si trabajamos simultáneamente con la *Nexys*, que no tiene lógica negada.

En la figura también se indican los pines de la FPGA que se conectan a los pulsadores, interruptores y LED. Como se puede apreciar, los pulsadores no tienen circuitería de anulación de rebotes[9], por lo que habrá un transitorio antes de recibir el valor definitivo.

[9] La anulación de rebotes se explica en el manual [17mach]

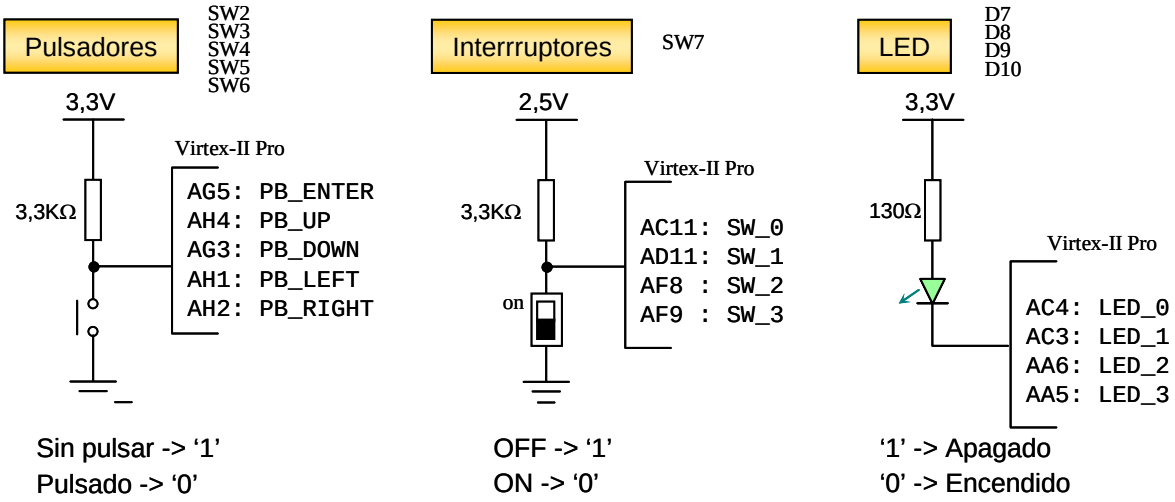


Figura 3.2: Conexiones de los pulsadores, interruptores y LED de propósito general en la placa XUPV2P

3.1.2. La tarjeta Nexys2

La tarjeta *Nexys2* [21nexys] se parece a la tarjeta *Basys* [3basys] (explicada en la referencia [17mach]). En la figura 3.3 se muestran algunos de sus componentes más importantes.

La *Nexys2* contiene una FPGA de Xilinx modelo *Spartan-3E XC3S500E* ó *XC3S1200E* con encapsulado FG320. Estas FPGA tienen respectivamente 500 000 ó 1 200 000 puertas lógicas equivalentes.

La placa *Nexys2* tiene algunos periféricos similares a la *XUP2P*, por ejemplo: 8 LED (XUPV2P tiene 4), 4 pulsadores (XUPV2P tiene 5), 8 interruptores (XUPV2P tiene 4), un puerto PS/2 (XUPV2P tiene 2), un puerto VGA (XUPV2P tiene uno de mejores características) y un puerto de alta velocidad. Ambas tienen memoria pero de distinto tipo.

La XUPV2P tiene puerto para conectar Ethernet, una tarjeta CompactFlash, audio y SATA, que son puertos que no tiene la *Nexys2*. Por otro lado, la *Nexys2* tiene cuatro *displays* de siete segmentos y conectores PMOD[10].

[10] A los conectores PMOD se le pueden conectar dispositivos 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

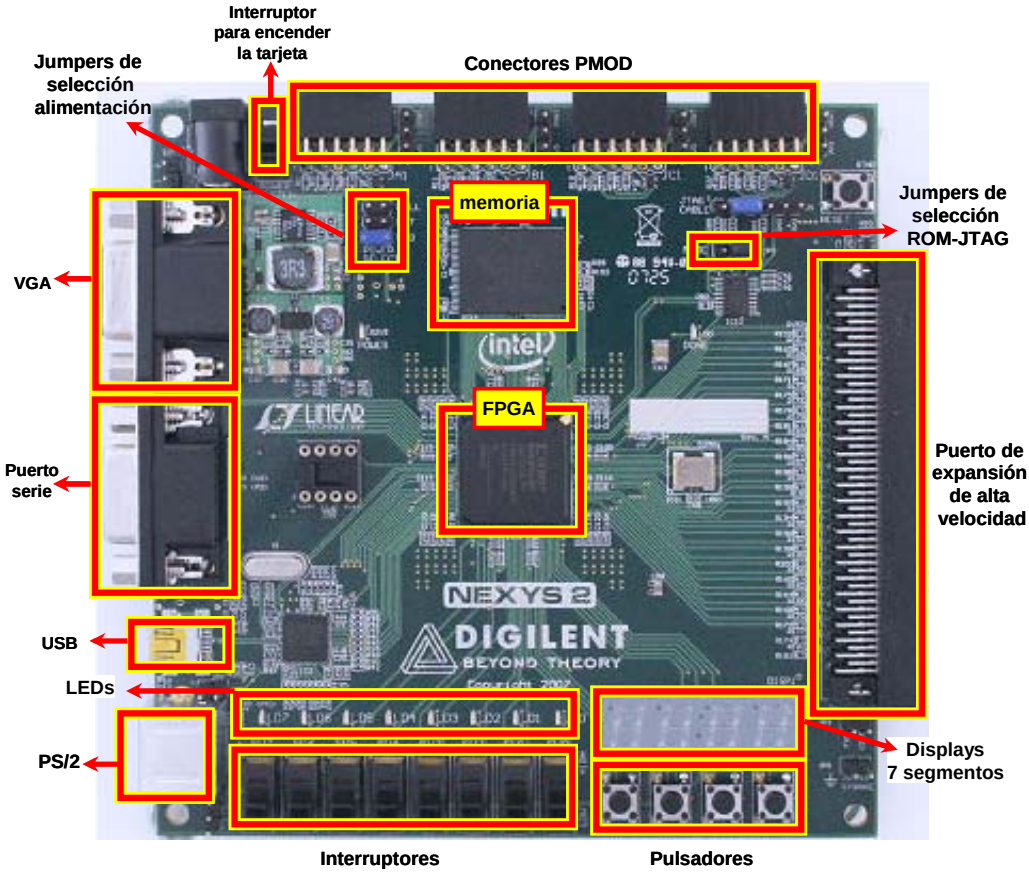


Figura 3.3: Placa Nexys2 con la que de manera alternativa se puede seguir este manual

En la figura 3.4 se muestran las conexiones en la placa de los pulsadores, interruptores y LED. Analizando la figura podemos observar que al pulsar o conectar un interruptor enviamos un '1' al pin de la FPGA, y de manera análoga, los LED se encienden al poner un '1' en el pin de la FPGA correspondiente. Por lo tanto, **en la tarjeta Nexys2 se trabaja con lógica directa con los pulsadores, interruptores y LED**. Esto es lo contrario que ocurre con la XUPV2P (recuerda la figura 3.2).

Como se puede apreciar, los pulsadores tampoco tienen circuitería de anulación de rebotes. En la figura también se indican los pines de la FPGA que se conectan a éstos.

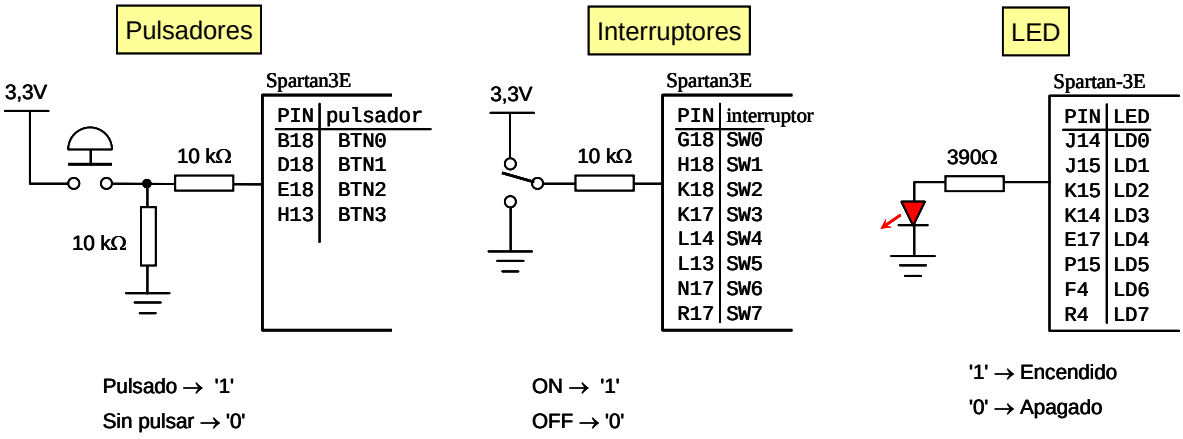


Figura 3.4: Conexiones de los pulsadores, interruptores y LED de propósito general en la placa Nexys2

3.2. Encender los leds

En esta primera práctica se realizará un circuito muy simple para asegurarnos de que funciona el proceso de programación de la FPGA y la placa, y así descartar posibles fallos del diseño. Los leds nos mostrarán si nuestro circuito funciona correctamente. Aunque esta práctica se explica con cierto detalle, si ves que te cuesta seguirla te recomendamos que consultes antes la referencia [17mach], donde se explica con más profundidad todo el proceso.

Para este ejemplo se detallará el proceso de diseño con la herramienta *ISE* de *Xilinx*, usando la versión 9.2 del *ISE WebPACK*[11].

Para comenzar crearemos un proyecto nuevo, primero arrancamos el programa *Xilinx Project Navigator*, que se encuentra en *Inicio→Todos los Programas→Xilinx ISE 9.2i→Project Navigator*.

Nos puede aparecer una ventana con el *Tip of the day* que son indicaciones que hace la herramientas cada vez que la arrancamos. Pinchamos en *OK*, con lo que se cierra dicha ventana.

Normalmente la herramienta abre el último proyecto con que se ha estado trabajando, si es así, lo cerramos pinchando en: *File→Close Project*.

Para crear el nuevo proyecto, se pincha en *File→New Project.* y nos saldrá la ventana *New Project Wizard – Create New Project* como la mostrada en la figura 3.5. En ella pondremos el nombre del proyecto, que lo llamaremos *led1*, indicamos la ruta donde guardaremos el proyecto (*Project location*), que será `C:/practicas/dcse/tunombre`. Respecto al nombre y a la ruta, es conveniente:

- **No trabajar desde un dispositivo de memoria USB[12].**
- **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[13]. Una vez rellenados los tres recuadros pinchamos en *Next*.

[11] Puede ser que alguna de las ventanas mostradas sean de la versión 8.2. Esta diferencia no debería de ser transcendente

[12] 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.

[13] HDL: del inglés: *Hardware Description Language*

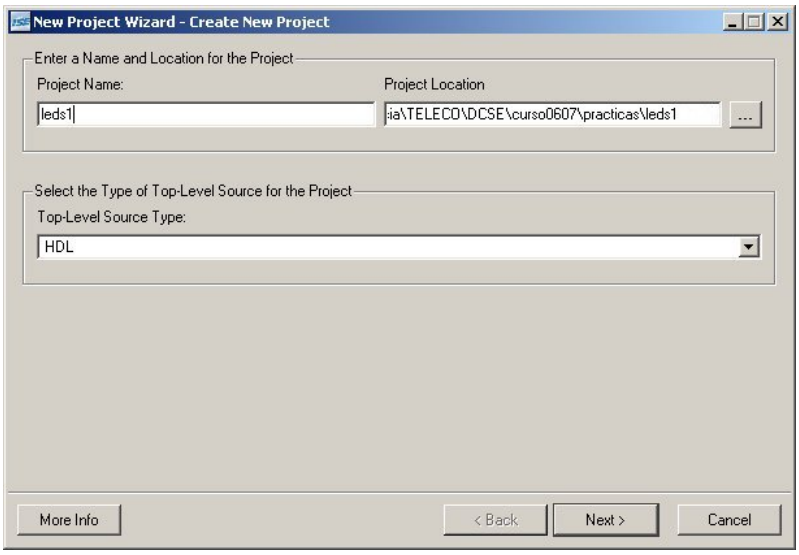


Figura 3.5: Ventana para crear un nuevo proyecto

A continuación aparecerá otro cuadro de diálogo en el que se pregunta por el tipo de dispositivo que se va a usar y por las herramientas que se van a utilizar a lo largo del flujo de diseño. Según la placa que utilicemos, la FPGA va a ser distinta, así que tendremos que distinguir si utilizamos la XUPV2P o la Nexys2. En las figura 3.6 y 3.7 se muestra cómo interpretar las características de las FPGA de las placas XUPV2P y Nexys2 a partir de sus encapsulados.

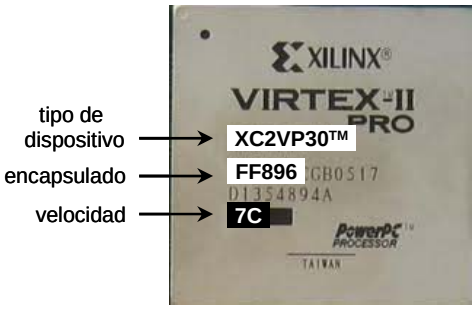


Figura 3.6: Interpretación del texto del encapsulado de la FPGA de la XUPV2P

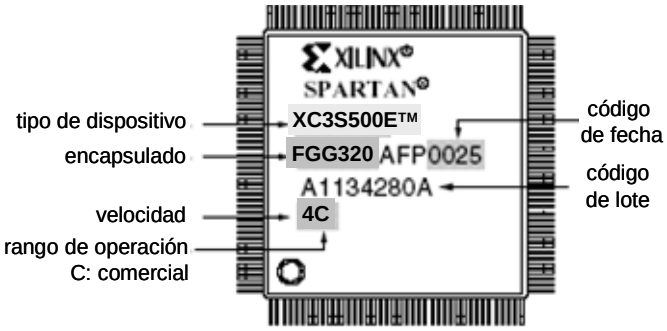
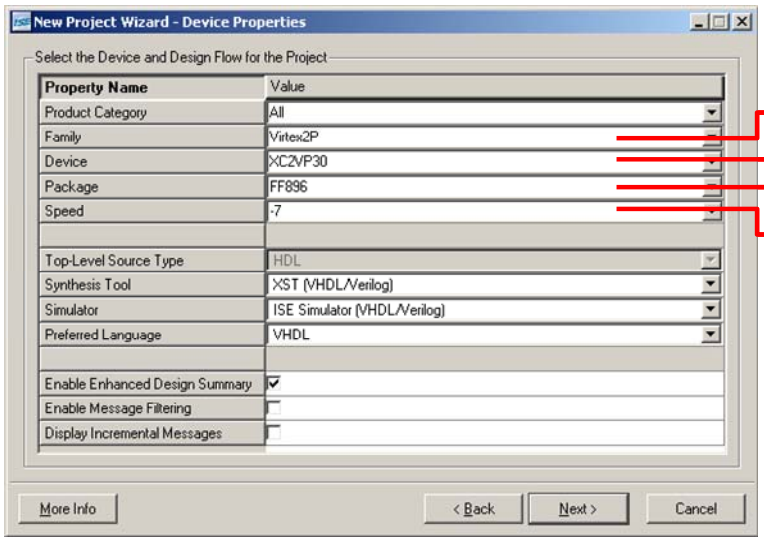


Figura 3.7: Interpretación del texto del encapsulado de la FPGA de la Nexys2

A partir de la información de los encapsulados de las FPGA tendremos que rellenar la ventana de la figura 3.8.

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)*, aunque si lo tienes instalado, podrías poner el *Modelsim*. En el lenguaje de descripción de hardware preferido (*Preferred Language*) se deja el *VHDL*, que es el que usaremos. Al terminar pinchamos en *Next*.



	XUPV2P	Nexys2
Family	Virtex2P	Spartan3E
Device	XC2VP30	XC3S500E
Package	FF896	FG320
Speed	-7	-4

Figura 3.8: 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* o haciendo doble clic en *Create New Source*, que se encuentra en la subventana *Processes*.

Con esta aparecerá una nueva ventana que nos pedirá el tipo de fuente que queremos crear. Como estamos trabajando en VHDL, seleccionamos *VHDL Module*, y nombramos al fichero *led1*. Este paso se muestra en la figura 3.9, a continuación pinchamos en *Next*.

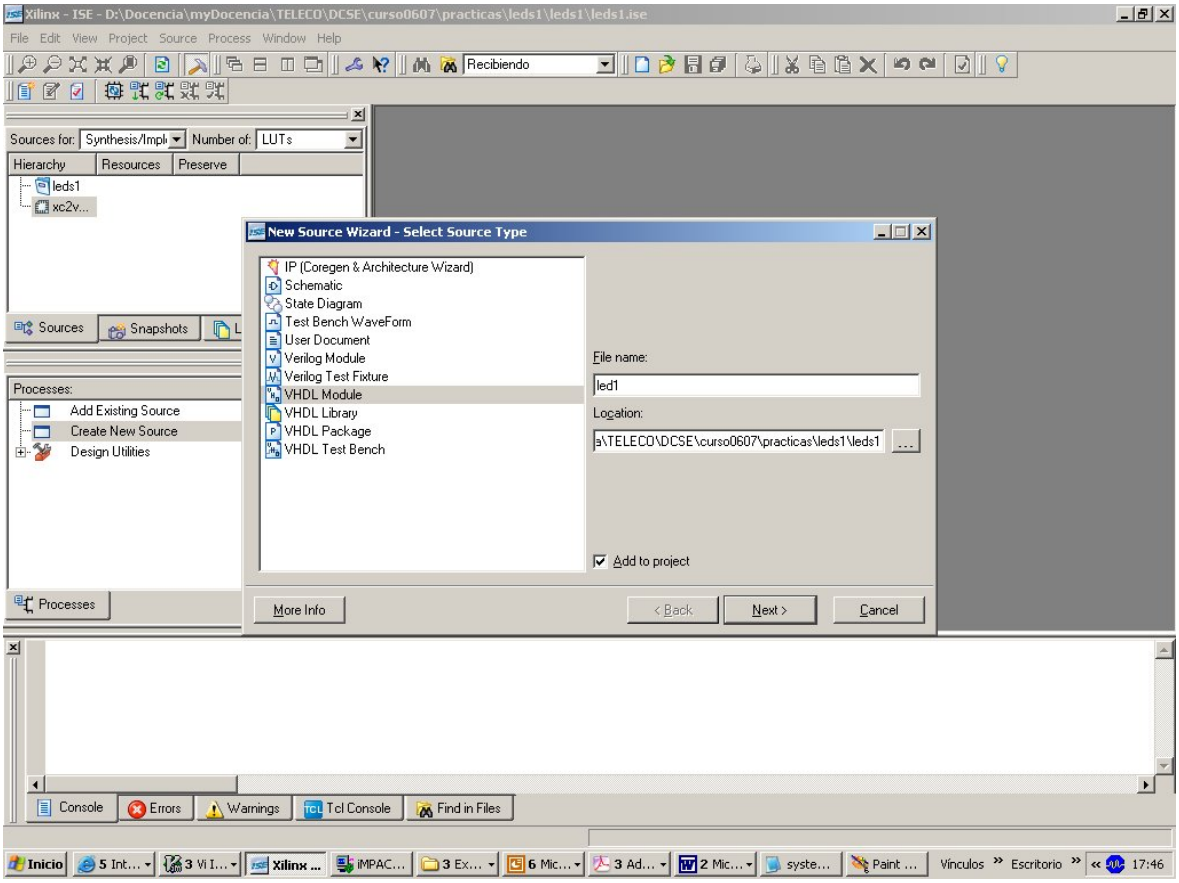


Figura 3.9: Selección del tipo de la nueva fuente que vamos a crear

Ahora nos saldrá una ventana como la mostrada en la figura 3.10. 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 3.10 se han creado cinco puertos de entrada[14] (`PB_ENTER`, `PB_UP`, `PB_RIGHT`, `SW_0`, `SW_1`) y cuatro de salida (`LED_0`, `LED_1`, `LED_2`, `LED_3`). Otra alternativa es crear los puertos directamente en VHDL. Para terminar pinchamos en *Next* y luego en *Finish*.

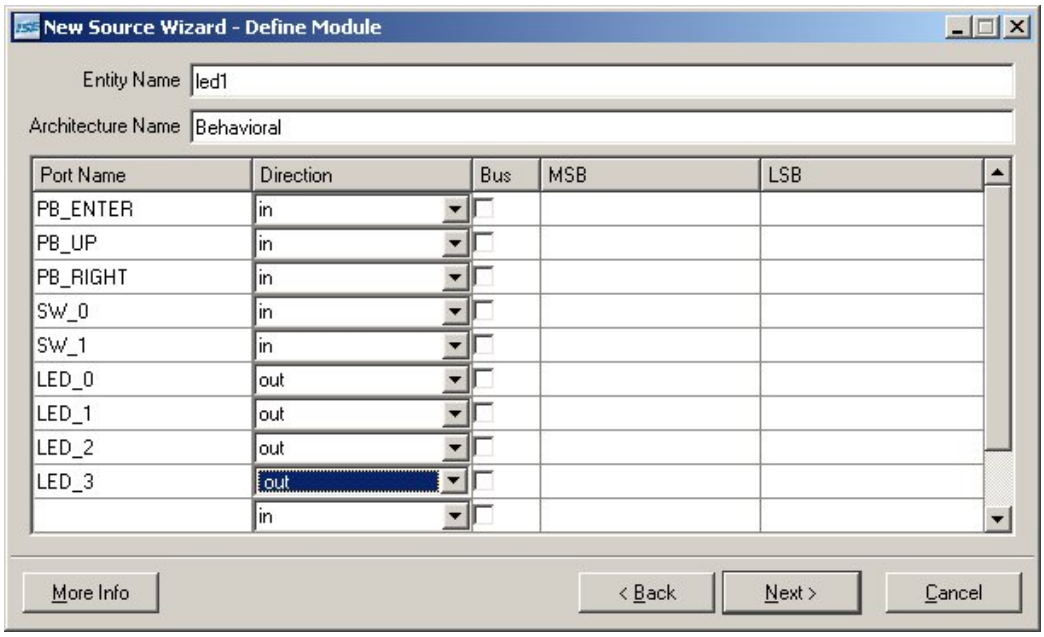


Figura 3.10: Definición de los puertos

Si al terminar el proceso la herramienta nos muestra la siguiente imagen (figura 3.11), debemos seleccionar el fichero que hemos creado (`led1.vhd`), ya sea pinchando en su pestaña, o en *Window→led1.vhd*.

[14] Los puertos se han nombrado siguiendo los nombres de los pulsadores de la placa *XUPV2P*, en caso de que estés usando la placa *Nexys2*, puedes nombrar los pulsadores como `PB_0` (en vez de `PB_ENTER`), `PB_1` (en vez de `PB_UP`) y `PB_2` (en vez de `PB_RIGHT`). El resto mantiene sus nombres.

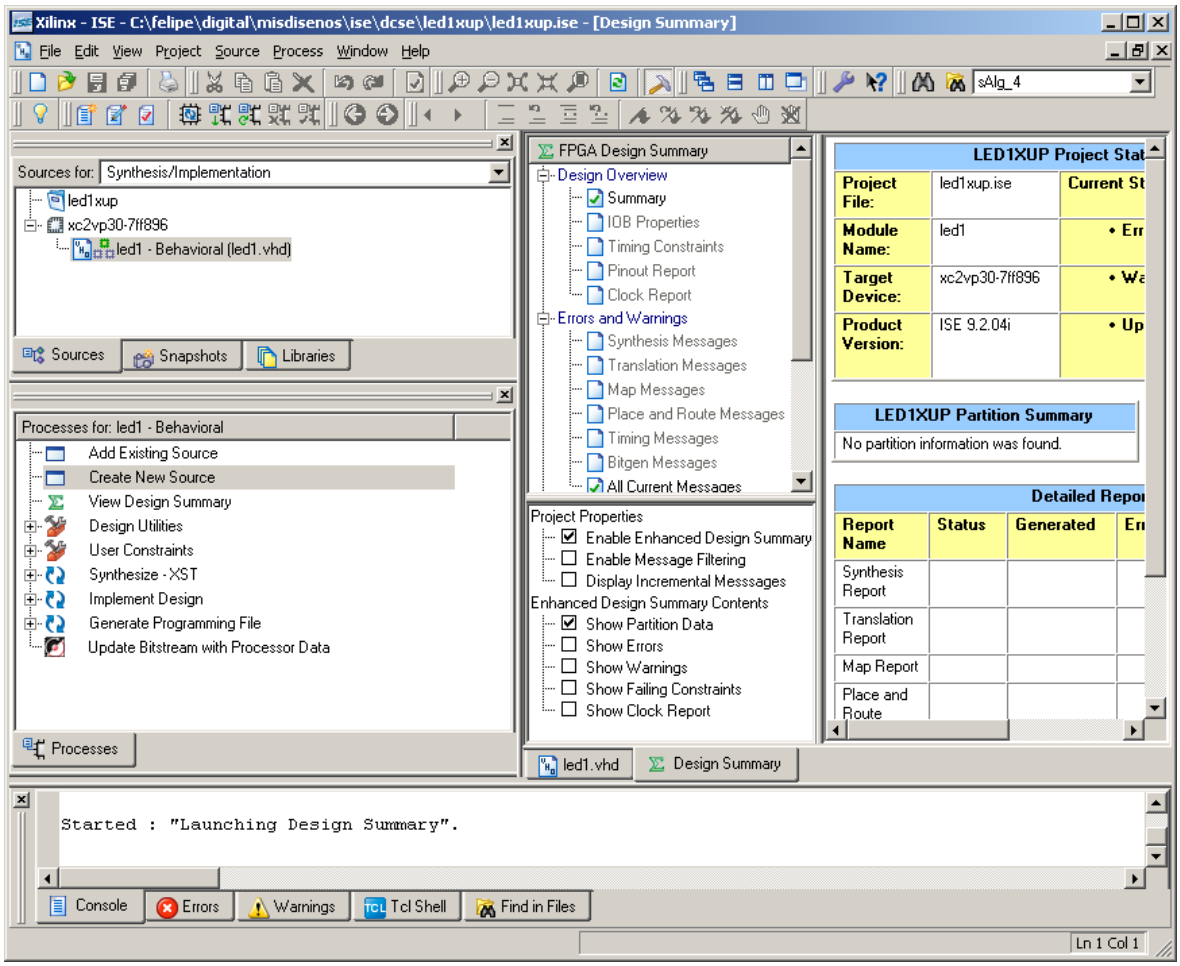


Figura 3.11: Apariencia de la herramienta al añadir la nueva fuente led1.vhd

Como muestra la figura 3.12, podemos ver que no sólo nos ha creado la entidad, sino que se han incluido unas cabeceras, bibliotecas, paquetes y la declaración de la arquitectura.

La cabecera es un apartado con comentarios para que lo rellenemos nosotros, para que tengamos un control del autor, la fecha y la descripción del diseño, entre otra información. Recuerda que en VHDL los comentarios se identifican con dos guiones "--". Es interesante que lo rellenes, especialmente si trabajas en grupo. Pero incluso si trabajas solo es conveniente rellenarlo porque es común olvidarse de lo que se hace una vez que ha transcurrido cierto tiempo.

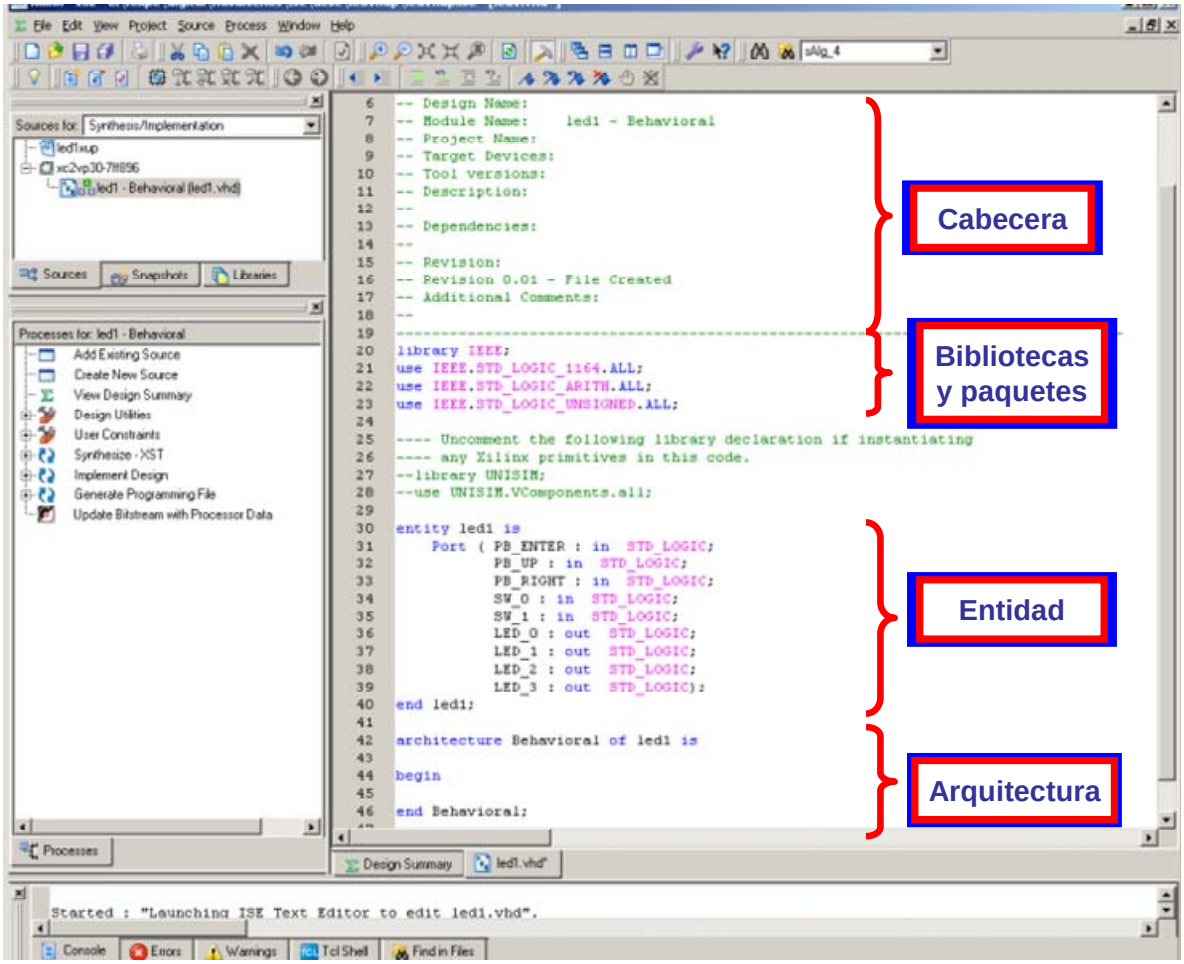


Figura 3.12: Apariencia de la herramienta al añadir la nueva fuente led1.vhd

Después de la cabecera se incluyen las referencias a los paquetes de la biblioteca[15] del IEEE[16], que son necesarias para poder utilizar los tipos de datos que emplearemos. En este ejemplo nos basta con utilizar el paquete `IEEE.STD_LOGIC_1164`. Los otros paquetes no son necesarios en este ejemplo.

En general, recomendamos utilizar el paquete `NUMERIC_STD` (código 3.2) en vez el `STD_LOGIC_ARITH` (código 3.1). No es obligatorio usar la `NUMERIC`, pero lo importante es usar siempre las mismas en un mismo diseño. En el capítulo 8 se profundiza en estos paquetes.

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

Código 3.1: Paquetes por defecto que pone el ISE

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

Código 3.2: Paquetes recomendados

Después de las referencias a las bibliotecas y paquetes hay unos comentarios que puedes quitar. A continuación está la **entidad**. La entidad tiene el aspecto mostrado en el código 3.3 o el código 3.4 según estemos usando la placa XUPV2P o la Nexys2 (si hemos cambiado los nombres de los puertos como decía la nota la pie 14).

[15] También llamadas librerías por su parecido con el término original inglés *library*, que realmente significa biblioteca. En inglés una librería se traduce como *bookstore*.

[16] El IEEE (Institute of Electrical and Electronics Engineers) es la asociación que ha estandarizado el VHDL.

```
entity led1 is
  Port ( PB_ENTER :  in  STD_LOGIC;
         PB_UP :     in  STD_LOGIC;
         PB_RIGHT :  in  STD_LOGIC;
         SW_0 :      in  STD_LOGIC;
         SW_1 :      in  STD_LOGIC;
         LED_0 :     out  STD_LOGIC;
         LED_1 :     out  STD_LOGIC;
         LED_2 :     out  STD_LOGIC;
         LED_3 :     out  STD_LOGIC);
end led1;
```

Código 3.3: Código de la entidad para la XUPV2P

```
entity led1 is
  Port ( PB_0 :    in  STD_LOGIC;
         PB_1 :    in  STD_LOGIC;
         PB_2 :    in  STD_LOGIC;
         SW_0 :    in  STD_LOGIC;
         SW_1 :    in  STD_LOGIC;
         LED_0 :   out  STD_LOGIC;
         LED_1 :   out  STD_LOGIC;
         LED_2 :   out  STD_LOGIC;
         LED_3 :   out  STD_LOGIC);
end led1;
```

Código 3.4: Código de la entidad para la Nexys2

La representación esquemática de estas entidades se muestra en las figuras 3.13 y 3.14. La entidad define las entradas y salidas del circuito, pero no hace referencia a cómo funciona o su estructura interna. Es como si fuese una caja negra.

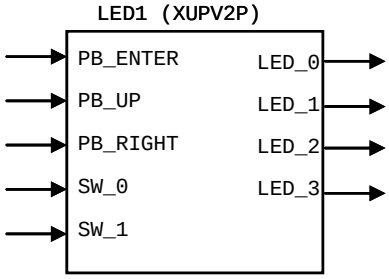


Figura 3.13: Esquema de la entidad para la XUPV2P

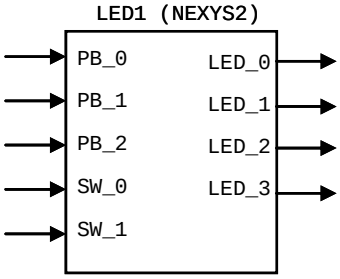


Figura 3.14: Esquema de la entidad para la Nexys2

Para describir el funcionamiento o la estructura interna del circuito se utiliza la arquitectura. En la figura 3.12 vemos que solo aparece la definición de la arquitectura, de modo que, como es lógico, el resto tenemos que describirlo nosotros.

Para este ejemplo, después del `begin` de la arquitectura, incluiremos el código 3.5 (si estás usando la *Nexys2* sigue el código 3.6):

```
 LED_0 <= '0';
 LED_1 <= '1';
 LED_2 <= PB_ENTER and SW_0;
 LED_3 <= PB_UP when (SW_1 = '1') else
          PB_RIGHT;
```

Código 3.5: Código de la arquitectura de led1.vhd en la XUPV2P

```
 LED_0 <= '0';
 LED_1 <= '1';
 LED_2 <= PB_0 and SW_0;
 LED_3 <= PB_1 when (SW_1 = '1') else
          PB_2;
```

Código 3.6: Código de la arquitectura de led1.vhd en la Nexys2

Una vez que hayas incluido el código 3.5, guarda el fichero y comprobamos si hemos tenido algún error de sintaxis pinchando en *Synthesize - XST→Check Syntax*. Esto está en la subventana de *Processes*. Observa la figura 3.15 para localizarlo.

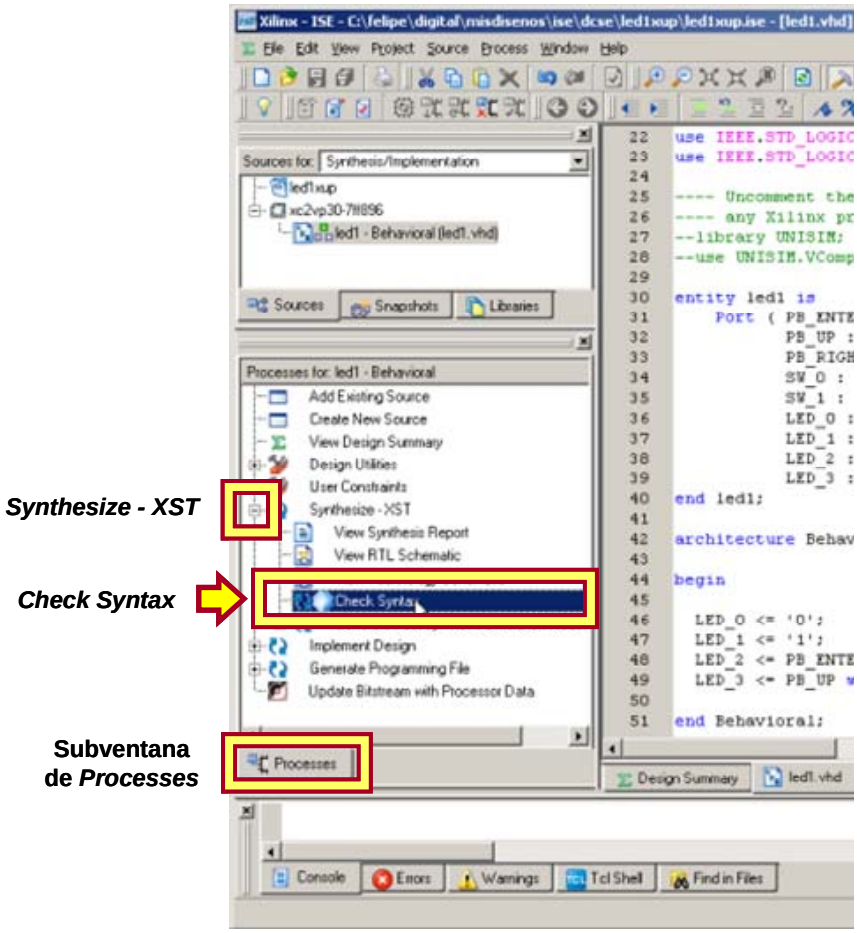


Figura 3.15: Comprobar la sintaxis: Synthesize - XST→Check Syntax

Si tienes algún error, localízalo y corrígelo. Cuando no tengas errores, puedes pinchar directamente en *Synthesize - XST* para sintetizar.

Una vez que hayamos sintetizado tenemos que indicar los pines de la FPGA que vamos a utilizar y su correspondencia con los puertos de la entidad.

En la ventana de *fuentes* (*Sources*) verifica que el componente y el fichero que hemos creado está seleccionado: *led1-Behavioral (*`led1.vhd`*)*. Pincha en él una vez si no estuviese seleccionado. 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 *Processes*, en ella despliega la sección que dice *User Constraints*, pincha dos veces en *Assign Package Pins*. 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*.

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 3.16).

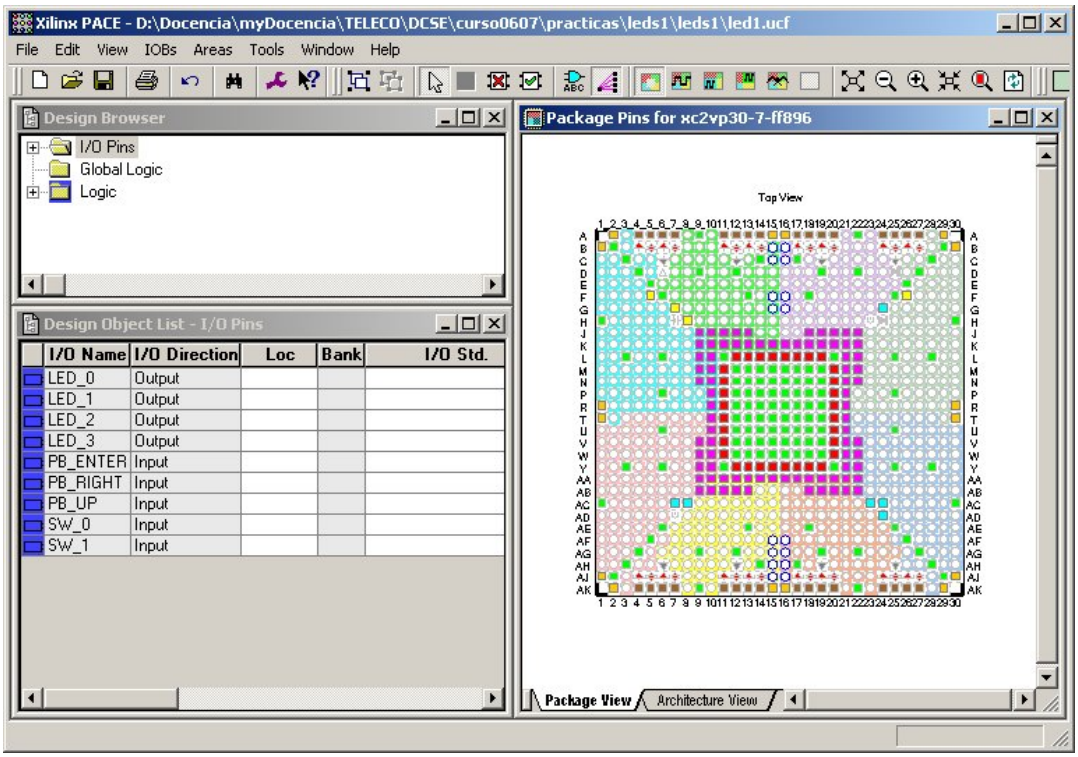


Figura 3.16: Herramienta PACE

A la izquierda, en la ventana *Design Object List* están los puertos de entrada salida[17]. Y que asignaremos según se mostró en la figura 3.2 para la *XUPV2P* o figura 3.4 para la *Nexys2*. En la 3.17 se muestra la asignación de los pines para la *XUPV2P*. Aunque en la columna *Loc* salga un menú desplegable con los bancos, es un error y tienes que introducir el pin correspondiente con el teclado. La columna de los *Bancos* (*Bank*) no tienes que rellenarla, la rellena automáticamente la herramienta al introducir el pin.

Para saber el código de los pines puedes consultar los manuales de referencia de las placas (referencias [31xup] y [21nexys]). En el caso de la *Nexys2*, algunos de los pines están impresos en la propia placa entre paréntesis, sin embargo, algunos de ellos son erróneos, como los de los pulsadores que los tiene intercambiados: El `BTN0` por el `BTN3`, y el `BTN1` por el `BTN2`.

[17] A veces no salen los puertos correctamente. Cuando esto ocurra revisa que no tengas ningún espacio o carácter extraño en la ruta de tu proyecto. Revisa también que el diseño se sintetiza correctamente.

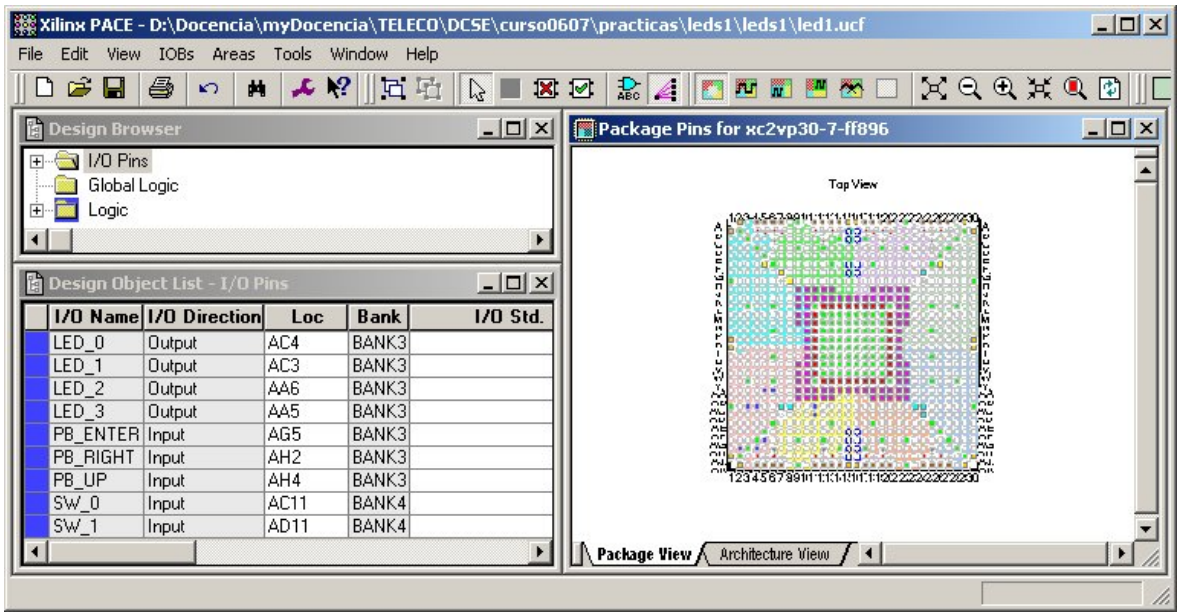


Figura 3.17: Asignación de los pines en la herramienta PACE para la XUPV2P

Los pines para la *Nexys2* serían los siguientes (figura 3.18).

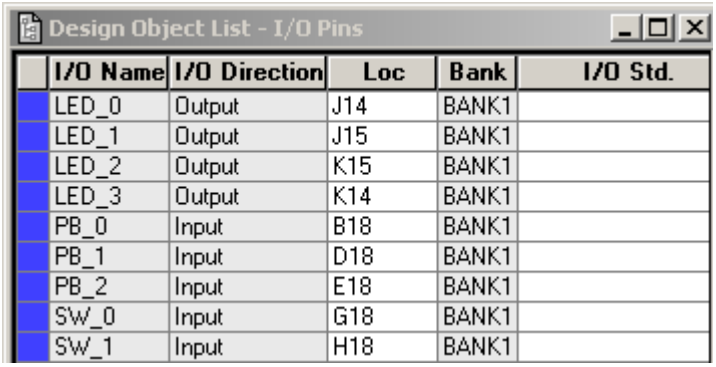


Figura 3.18: Asignación de los pines en la herramienta PACE para la Nexys2

Una vez que hayas incluido los pines, guardamos y cerramos la herramienta PACE. Seguramente nos salga un aviso que nos pregunta cómo queremos guardar el fichero `.ucf`. Para nosotros es indiferente y pondremos la primera de las opciones ("*XST Default <>*").

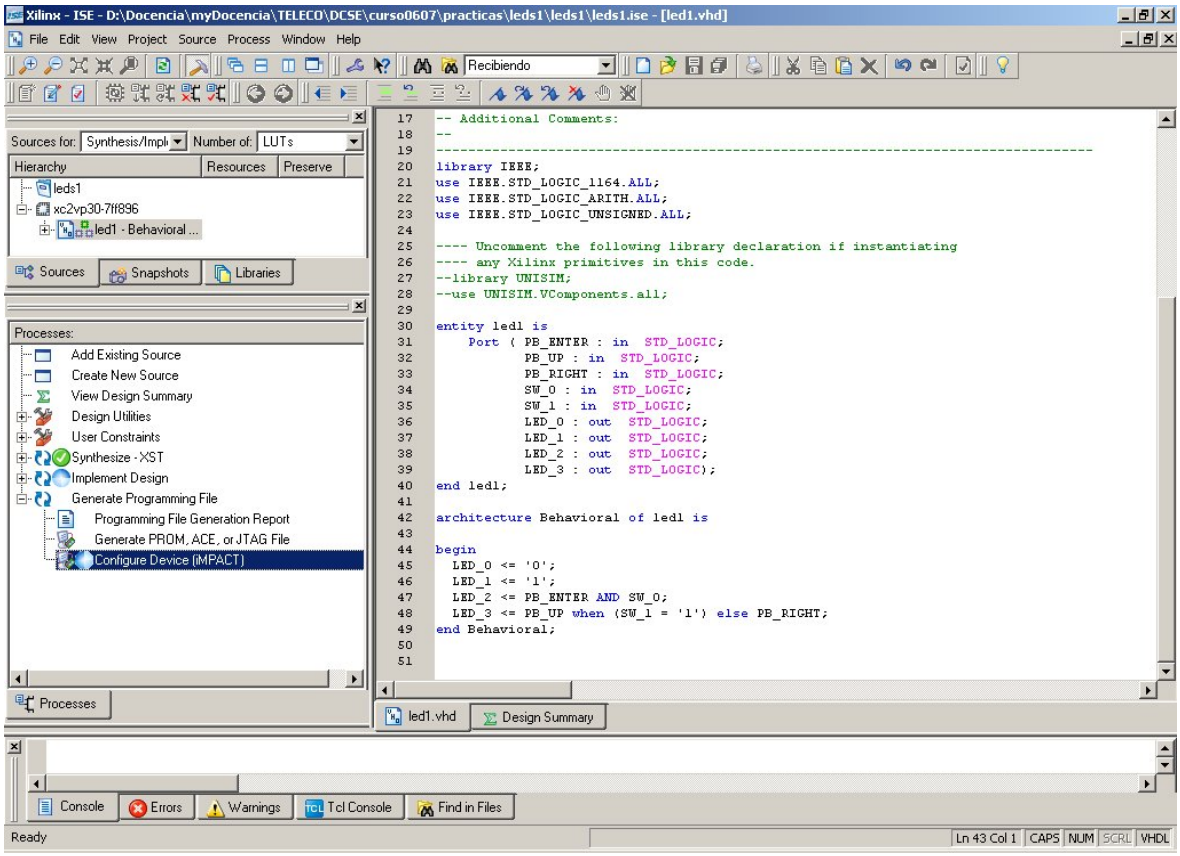
Ahora pinchamos dos veces en *Generate Programming File* dentro de la ventana *Processes*, y esperamos a que termine el proceso de síntesis e implementación, viendo si hay algún error o advertencia de importancia (*warning*).

Si todo ha ido bien ya podemos implementar el circuito en las placas. Como el proceso es diferente según estemos usando la *XUPV2P* o la *Nexys2*, cada una de ellas se explicará en un apartado diferente.

3.2.1. Implementación en la tarjeta *XUPV2P*

Antes de programar la placa ponemos el interruptor `SW9` (*CONFIG SOURCE*) de manera que seleccione JTAG (ponemos ambos en `OFF`). Enchufamos el cable de alimentación y encendemos la placa (`SW11`). El led verde *JTAG CONFIG* (D20) deberá estar encendido y el led D11 (`Error System ACE`) podrá estar parpadeando (esto no es un error).

Enchufamos el cable USB al PC y a la placa. Esto podrá provocar un aviso en nuestro PC que nos indicará acerca de un nuevo hardware detectado. Si es la primera vez que lo enchufamos, necesitará instalar unos *drivers* y para ello necesitaríamos ser administrador, en este caso es normal que vuelva a aparecer el mismo mensaje, y volvemos a decir que lo instale. En la herramienta *ISE* dentro de la subventana *Processes* pulsamos dos veces en *Generate Programming File → ConfigureDevice (iMPACT)*, ver la figura 3.19**.**




```
17  -- Additional Comments:
18  --
19  ----------------------------------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22  use IEEE.STD_LOGIC_ARITH.ALL;
23  use IEEE.STD_LOGIC_UNSIGNED.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 led1 is
31      Port ( PB_ENTER : in  STD_LOGIC;
32             PB_UP : in  STD_LOGIC;
33             PB_RIGHT : in  STD_LOGIC;
34             SW_0 : in  STD_LOGIC;
35             SW_1 : in  STD_LOGIC;
36             LED_0 : out  STD_LOGIC;
37             LED_1 : out  STD_LOGIC;
38             LED_2 : out  STD_LOGIC;
39             LED_3 : out  STD_LOGIC);
40  end led1;
41
42  architecture Behavioral of led1 is
43
44  begin
45    LED_0 <= '0';
46    LED_1 <= '1';
47    LED_2 <= PB_ENTER AND SW_0;
48    LED_3 <= PB_UP when (SW_1 = '1') else PB_RIGHT;
49  end Behavioral;
50
51
```

Figura 3.19: Llamada a la herramienta iMPACT para programar la FPGA de la XUPV2P

Con esto, internamente estamos llamando a la herramienta *iMPACT*, con la que vamos a programar la FPGA. Aparecerá una ventana en la que pinchamos en *Finish*.

Puede pasar que salga un mensaje de error diciendo que no encuentra el cable, en este caso hay que revisar la configuración del cable de comunicación y comprobar que los pasos se han seguido bien (placa encendida e interruptores correctamente posicionados). Pinchando en *Output → Cable Setup…* se debe seleccionar dentro de *Communication Mode* el *Platform Cable USB* y pinchar en *OK* (ver figura 3.20).

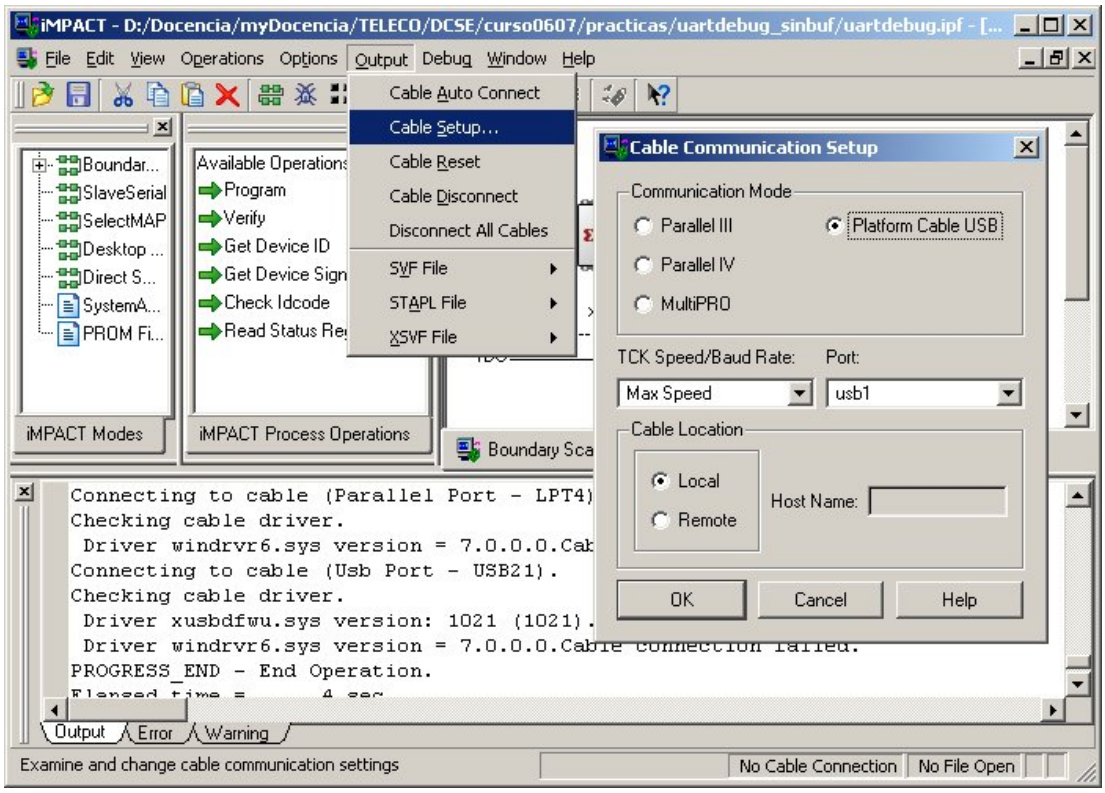


Figura 3.20: Configurando el cable USB para programar la FPGA de la XUPV2P

Si todo ha salido bien, saldrán tres ventanas para seleccionar un fichero y en todas seleccionamos *Cancel*. Veremos una ventana donde aparecen tres dispositivos, que indican los dispositivos JTAG que la herramienta ha identificado en la placa (figura 3.21).

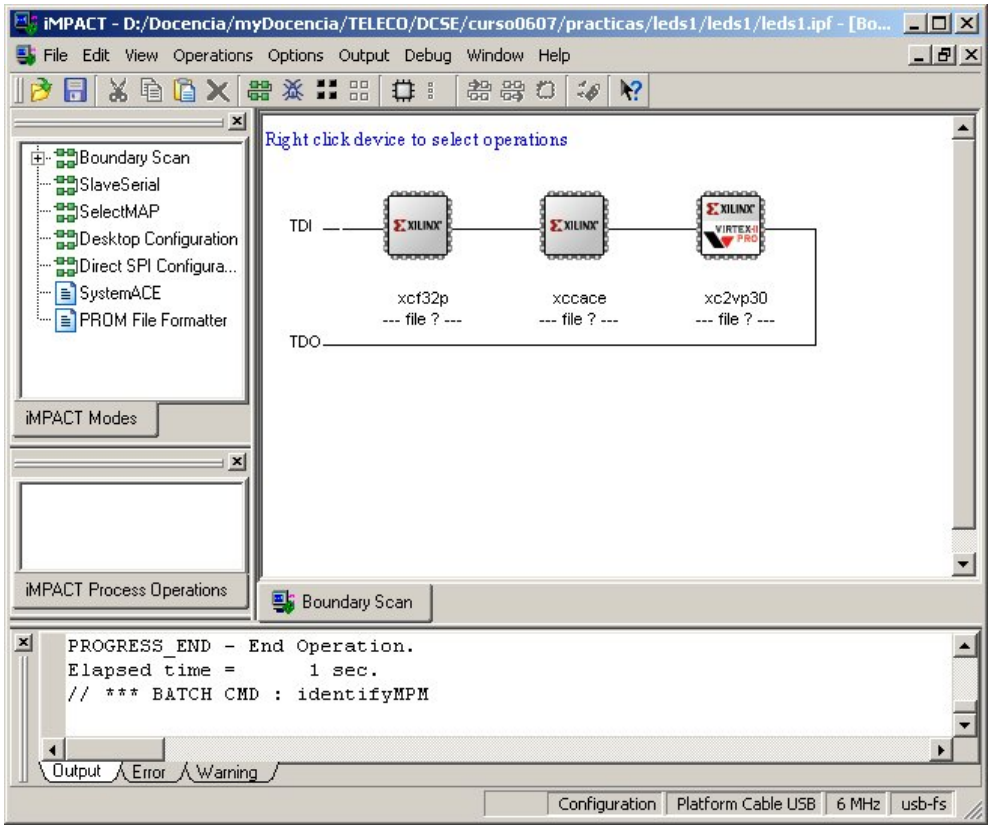


Figura 3.21: Dispositivos JTAG identificados en la XUPV2P

El último de la derecha es la FPGA que queremos programar (`xc2vp30`), sobre ella pinchamos con el botón derecho del ratón[18] y seleccionamos la opción *Assign New Configuration File* y escogemos el fichero `led1.bit`, que es el fichero de configuración de la FPGA con nuestro circuito sintetizado. Le damos a *OK* en la siguiente ventana.

Volvemos a pinchar con el botón derecho para indicarle que la queremos programar, seleccionamos *Programm...*, y en la ventana que aparece pinchamos en *OK* (sin poner la opción de verificar).

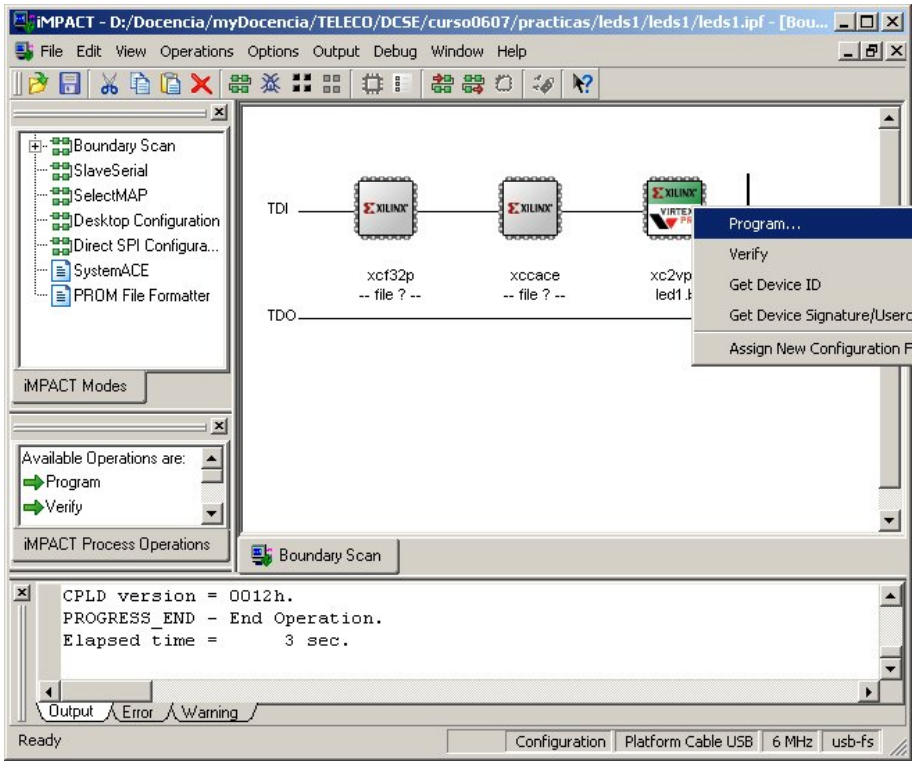


Figura 3.22: Programación de la FPGA de la XUPV2P

Por último, *iMPACT* nos indicará si ha realizado la programación correctamente (figura 3.23) o ha habido algún error (figura 3.24). En caso de que haya habido algún error puede ser que te hayas equivocado en el tipo de FPGA seleccionado consulta el apartado 3.3 para saber cómo se hace.

[18] A veces se atasca el ratón, si es así pincha con el botón izquierdo del ratón fuera de los tres dispositivos (en la parte blanca), después de esto seguramente ya puedas pinchar con el botón derecho sobre el dispositivo.

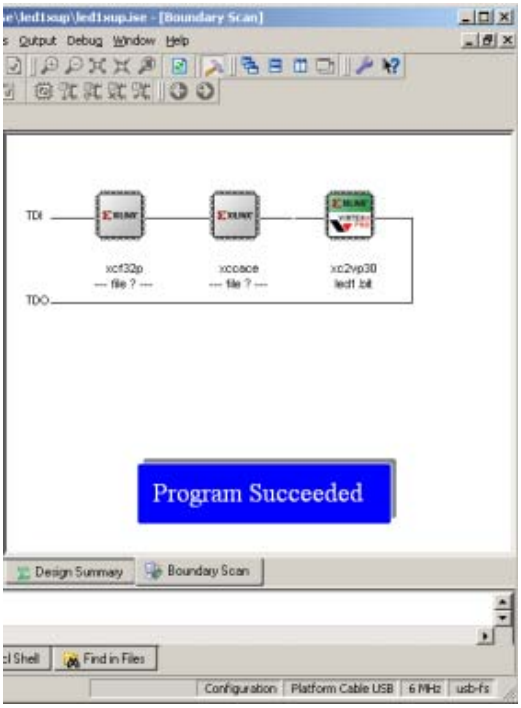


Figura 3.23: Programación exitosa de la FPGA

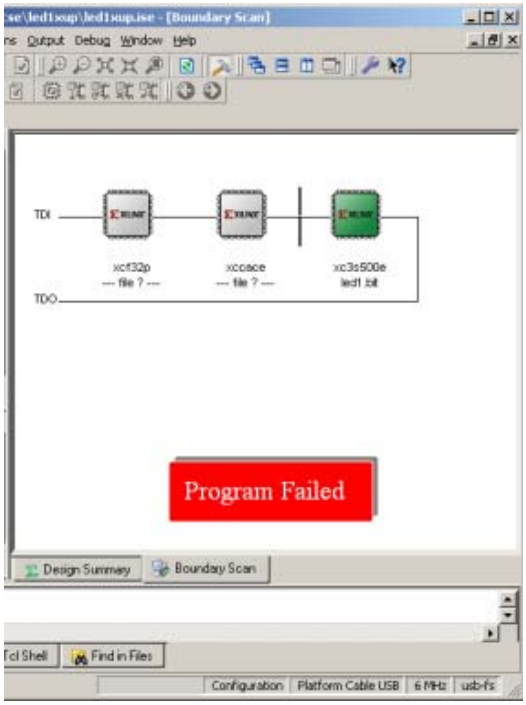


Figura 3.24: Programación fallida de la FPGA

Si todo ha salida En ese caso, se encenderá el led rojo `DONE` (debajo del verde que ya estaba encendido). Y se encenderá el `LED0`, ya que como le habíamos asignado un `'0'` hace que se encienda. El `LED1` estará apagado.

El `LED2 (LED_2 <= PB_ENTER and SW_0;)` estará apagado (`'1'`) cuando `SW_0='1' (OFF)` y `PB_ENTER='1'`. Presionando `PB_ENTER` o/y poniendo `SW_0` a `ON`, se nos encenderá el led. De manera parecida podemos ver si el `LED3` funciona cuando presionamos el `PB_UP` si `SW_1` está en `OFF`; y cuando presionamos `PB_RIGHT` si `SW_1` está en `ON`.

Fíjate que esto es así porque los pulsadores, interruptores y led de la XUPV2P trabajan con lógica negada.

3.2.2. Implementación en la tarjeta *Nexys2*

Aunque la tarjeta *Nexys2* 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* [1adept].

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 3.3 (*jumpers* de selección ROM-JTAG). 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*). 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.

Pon también los *jumpers* de *POWER SELECT* (figura 3.3: *jumpers* de selección alimentación) en la posición USB.

A continuación se explica cómo se programa la tarjeta Nexys2 con la versión 2 del *Adept*. Ejecuta el programa *Adept*: *Inicio→Todos los Programas→Digilent→Adept*. En la figura 3.25 se muestra la pantalla inicial cuando no hay ningún dispositivo conectado. Esto se puede ver en la parte derecha de la ventana.

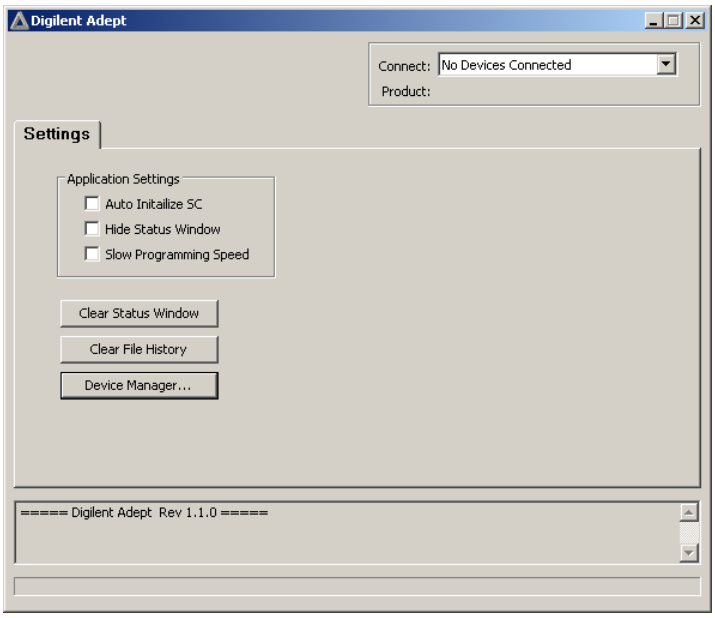


Figura 3.25: Pantalla inicial del Adept, sin dispositivos conectados

Ahora conectamos la tarjeta Nexys2 al ordenador mediante el cable USB y ponemos el interruptor de alimentación en la posición ON. Transcurrido un tiempo, en la ventana del *Adept* aparecerá que se ha conectado una tarjeta, en la parte derecha pondrá *Onboard USB* (ver figura 3.26). 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 tarjeta le proporciona los *drivers*).

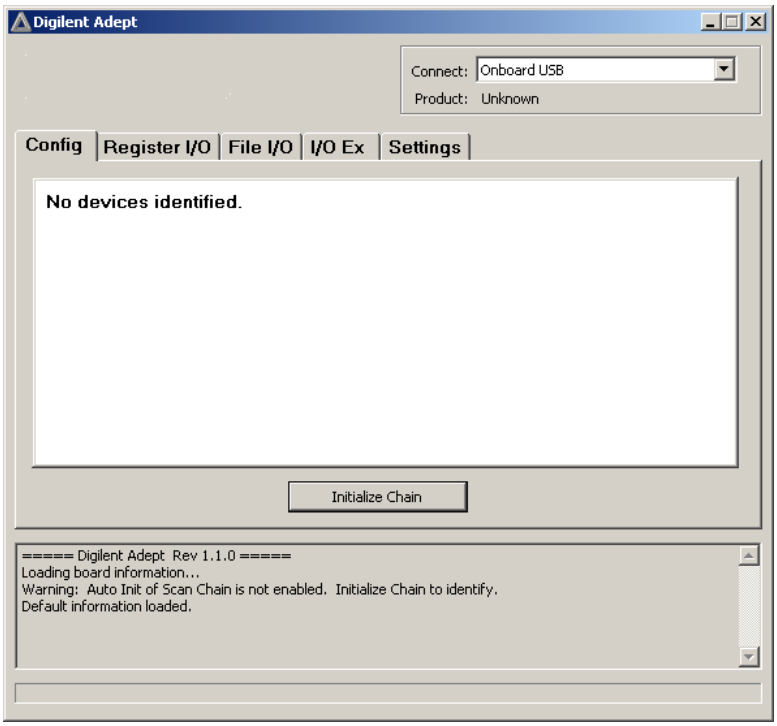


Figura 3.26: Pantalla del Adept que ha detectado algún dispositivo conectado

Ahora pinchamos en *Initialize Chain* y, como se ve en la figura 3.27, aparecerán dos componentes: uno es la FPGA y otro la memoria (PROM). Indicará también el modelo de FPGA, que en nuestro es la `XC3S500E`.

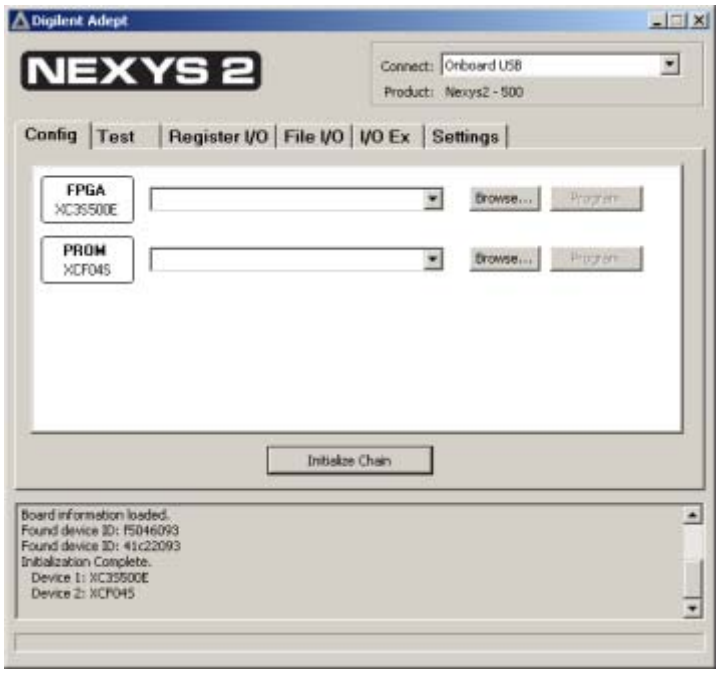


Figura 3.27: Pantalla del Adept que ha detectado la cadena JTAG de dispositivos (FPGA y PROM)

A continuación pinchamos en el botón *browse...* correspondiente a la FPGA, y buscamos el fichero `led1.bit` que hemos generado en nuestro proyecto. Aparecerá una advertencia (*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 3.3 si no sabes cómo se realiza esta operación.

Si todo ha salido bien, sólo tenemos que pinchar en ***Program*** y se programará la FPGA. El resultado será que se encenderá el led `LD1`, y el led `LD0` estará apagado. Observa que es justo lo contrario que ocurría en la XUPV2P. El comportamiento de los otros led es también diferente por tener lógica directa. Comprueba que funciona como debería ser.

3.3. Cambiar el tipo de FPGA de un proyecto

Este apartado se incluye por si en un proyecto queremos modificar el tipo de FPGA, ya sea porque vamos a cambiar de placa (por ejemplo de la *XUPV2P* a la *Nexys2*) o porque nos hemos equivocado al principio poniendo las características (recuerda la figura 3.8), en cualquier momento podemos realizar el cambio. Para ello, tenemos que seleccionar la FPGA en la subventana *Sources*, y pinchar con el botón derecho del ratón. En el menú desplegable que aparece tendremos que pinchar en *Properties* (figura 3.28). Tras esta operación aparecerá la ventana de la figura 3.8 que nos permitirá cambiar el modelo y las características de la FPGA.

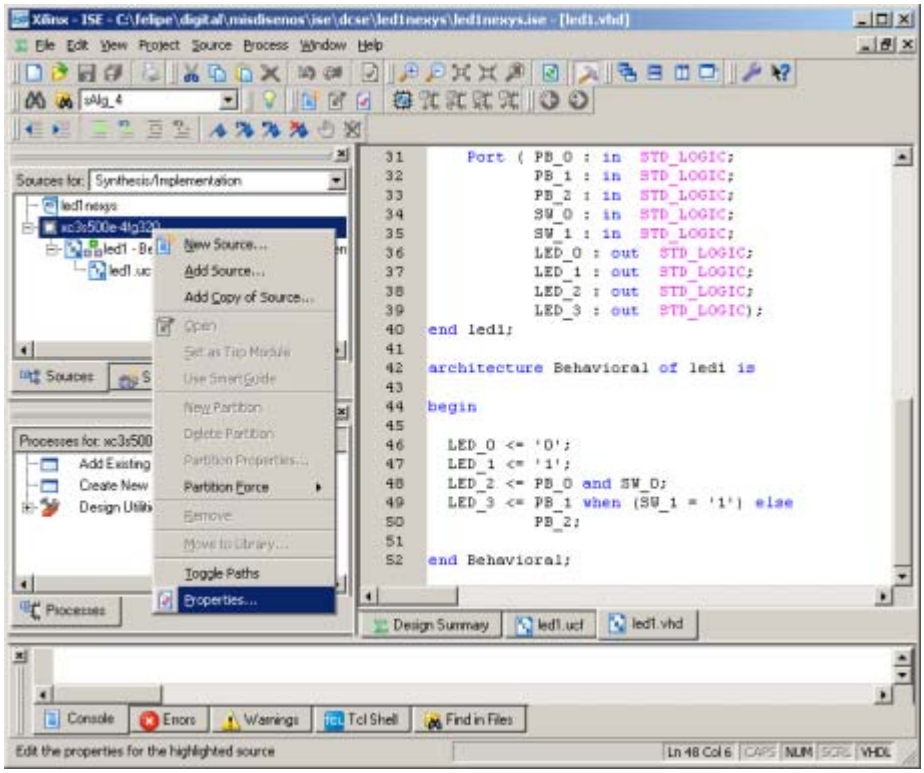


Figura 3.28: Procedimiento para cambiar las características de la FPGA

Ten en cuenta que según el cambio que hayamos realizado, quizá 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). Otras cosas que podemos necesitar cambiar son otras características del funcionamiento: la frecuencia de

reloj puede haber cambiado, la lógica de los pulsadores, periféricos,... por tanto a veces el cambio no es tan fácil.

3.4. Las restricciones del circuito

Hemos visto que dentro de la subventana *Processes* está el apartado *User Constraints*. En donde, mediante la herramienta *PACE* hemos asignado los pines de la FPGA. En el apartado 4.1 veremos que también se pueden poner restricciones temporales.

Gran parte de esta información se guarda en un fichero de texto, con extensión `.ucf`, que se puede visualizar y modificar con cualquier editor de texto, y también se puede editar directamente desde la herramienta *ISE* pinchando dentro de la subventana *Processes* en *User Constraints* → *Edit Constraints (Text)*. Ver figura 3.29.

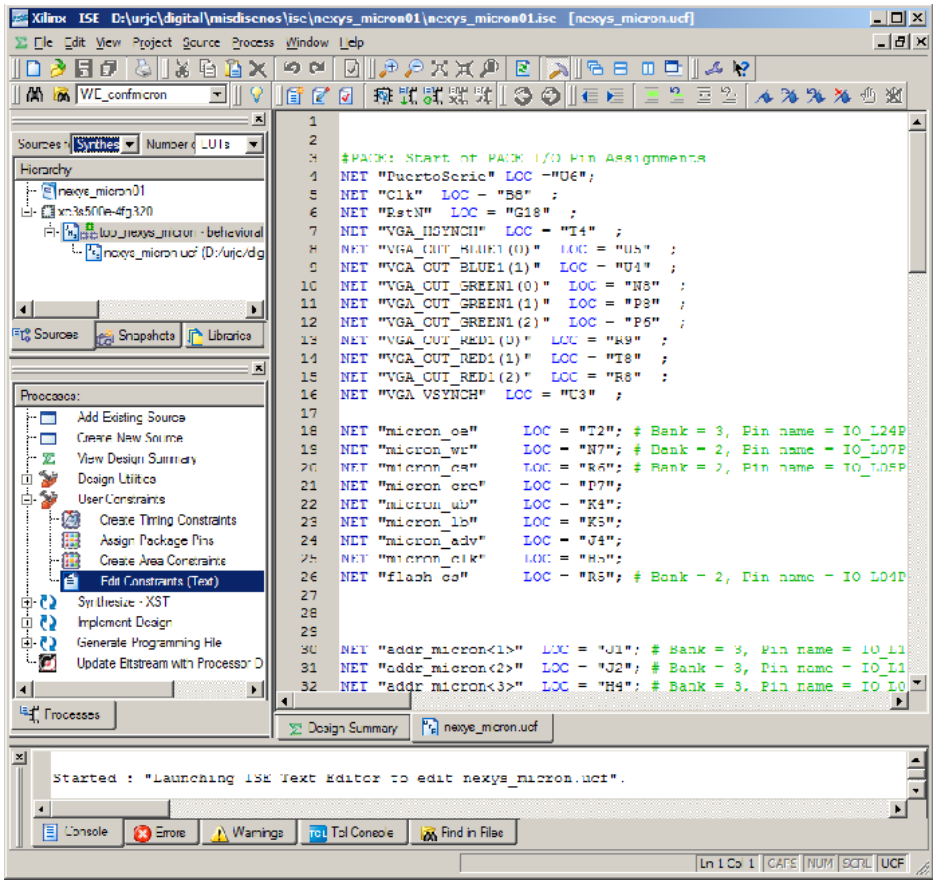


Figura 3.29: Edición del fichero de restricciones

En el fichero podemos ver la asociación de puertos (`NET`) con pines (`LOC`). Esta información se extrae de la guía de usuario de la placa. Como estamos trabajando con una placa ya fabricada, los pines tienen una localización fija en la placa, y por tanto siempre que usemos los mismos pines podemos usar el mismo fichero `.ucf`. En la página web de la placa están disponibles estos ficheros para que simplemente tengamos que copiar y pegar en nuestro diseño[19].

Para el caso de la *XUPV2P*, dentro del fichero comprimido hay distintos ficheros `.ucf` agrupados por la funcionalidad de los pines (botones, leds, puerto serie,...). En estos

[19] De la XUPV2P: http://www.xilinx.com/univ/XUPV2P/UCF_Files/UCF_FILES.zip
De la Nexys2: http://www.digilentinc.com/Data/Products/NEXYS2/Nexys2_500General.zip

ficheros hay más información, que indica el tipo de lógica de cada pin (`LVTTL`, `LVCMOS`,...) y, en el caso del .ucf de los relojes (`CLOCKS.UCF`) la frecuencia de cada reloj.

Por tanto, para evitar equivocarnos al asignar los pines, en vez de utilizar la herramienta *PACE* podemos copiar y pegar de los ficheros .ucf. Eso sí, los puertos deben llamarse igual que en el fichero .ucf y si no, cambiarlos nosotros con el editor.

4. Utilización del reloj y circuitos secuenciales

En la práctica anterior diseñamos un circuito combinacional, que no tenía reloj. Ahora realizaremos una práctica con un circuito secuencial sencillo para recordar cómo se hace. Esta es una práctica de repaso, si tienes dudas te recomendamos que mires el manual [17mach].

Los ejemplos serán sencillos y servirán para repasar. También comprobaremos que los relojes de las placas funcionan bien y nos ayudará a entender las diferencias entre las dos placas.

4.1. Segundero (primera versión)

En este ejercicio haremos que uno de los led parpadee. El led estará un segundo apagado y el siguiente segundo permanecerá encendido. Para ello utilizaremos el reloj de la placa. Sin embargo, la **frecuencia del reloj** de la tarjeta *XUPV2P* **es diferente** a la de la *Nexys2*. El reloj de la *XUPV2P* va a 100 MHz mientras que el de la *Nexys2* va a 50MHz. Este es un dato muy importante a tener en cuenta en los diseños que tengas que realizar a lo largo del curso.

En la *XUPV2P*, el reloj entra por el pin `AJ15` de la FPGA, en la *Nexys2* el reloj lo tenemos en el pin `B8`.

Como seguramente recuerdes, para implementar un segundero podemos utilizar un divisor de frecuencia (un contador) que cada vez que llegue al final de su cuenta dará una señal de aviso de que ha pasado un segundo.

Además añadiremos una señal de reset que se corresponderá con un pulsador. En la *XUPV2P* utilizaremos el pulsador del medio (`ENTER`, pin: `AG5`), mientras que en la Nexys2 utilizaremos el pulsador de la derecha (`BTN0`, pin: `B18`).

Anteriormente hemos visto que los pulsadores de la *XUPV2P* envían un cero cuando están pulsados, por tanto, el reset sería a nivel bajo (se resetea cuando recibe un cero). Sin embargo en la *Nexys2*, los pulsadores envían un uno cuando están pulsados.

Los leds de las placas también funcionan de forma inversa, sin embargo, como van a estar la mitad del tiempo encendidos y la otra mitad apagados, probablemente no nos daremos cuenta.

Por lo tanto tenemos dos diferencias importantes al usar las placas:

- Tienen frecuencia de reloj diferente
- La lógica de los pulsadores, interruptores y leds es diferente (lógica directa y lógica negada).

Siguiendo los pasos de la práctica anterior, crearemos un nuevo proyecto llamado `SEG1`. En este caso sólo tendremos 3 puertos, dos de entrada (`clk`, `rst`) y uno de salida (`led0`). Sin embargo, debido a las diferencias entre las placas podría ser conveniente nombrar a los puertos de distinta manera para distinguir los diseños. Así, la señal de reloj de la *XUPV2P* la podemos llamar `clk_100mhz` y la de la Nexys2 la llamaríamos `clk_50mhz`.

Por otro lado, las señales de reset son diferentes, porque al pulsar el reset en la *XUPV2P* se recibe un cero, y en la *Nexys2* se recibe un uno. Por tanto, el reset de la *XUPV2P* es activo a nivel bajo, y eso se puede representar poniéndole una '_n' (de negado) al final de su nombre: `rst_n`.

En las figuras 4.1 y 4.2 se muestran las representaciones esquemáticas de estas entidades.

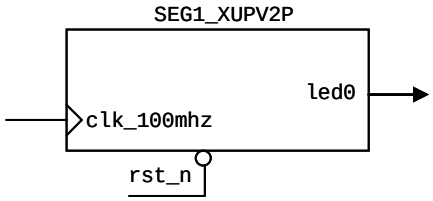


Figura 4.1: Esquema del segundero para la XUPV2P

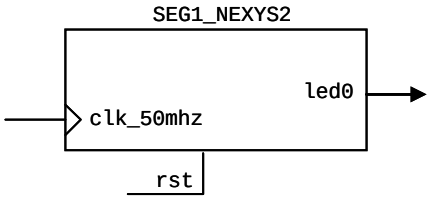


Figura 4.2: Esquema del segundero para la Nexys2

Si has realizado las prácticas de Electrónica Digital II [17mach] seguramente recuerdes cómo se hace un segundero. Lo ideal sería que intentases realizarlo por ti mismo. Aún así, a continuación se muestra un ejemplo de código para que lo implementes. Obsérvalo bien e intenta descubrir si es correcto, **ya que tiene algún fallo**. En el código 4.1 se muestra la versión para la *XUPV2P*.

```
entity SEG1_XUP is
  Port (
    rst_n       : in    std_logic;
    clk_100mhz  : in    std_logic;
    led0        : out   std_logic
  );
end SEG1_XUP;

architecture BEHAVIORAL of SEG1_XUP is
  signal contseg   : natural range 0 to 100000000; -- XUPV2P: 100MHz a 1 segundo
  signal led0aux   : std_logic;
begin
  led0 <= led0aux;
  P_Conta: Process (rst_n, clk_100mhz)
  begin
    if rst_n = '0' then    -- XUPV2P: reset activo a nivel bajo
      contseg <= 0;
      led0aux <= '0';
    elsif clk_100mhz'event and clk_100mhz='1' then
      contseg <= contseg + 1;
      if (contseg = 0) then
        led0aux <= not led0aux;
      end if;
    end if;
  end process;
end BEHAVIORAL;
```

*Código 4.1: Código de la primera versión (**con fallos**) del segundero para la XUPV2P*

Recuerda que se ha tenido que crear la señal auxiliar `led0aux` ya que los puertos de salida no se pueden asignar.

En el código 4.2 se muestra la versión (también con algún fallo) para la *Nexys2*.

```
entity SEG1_NEXYS is
  Port (
    rst       : in    std_logic;
    clk_50mhz : in    std_logic;
    led0      : out   std_logic
  );
end SEG1_NEXYS;

architecture BEHAVIORAL of SEG1_NEXYS is
  signal contseg   : natural range 0 to 50000000; -- Nexys2: 50MHz a 1 segundo
  signal led0aux   : std_logic;
begin
  led0 <= led0aux;
  P_Conta: Process (rst, clk_50mhz)
  begin
    if rst = '1' then  -- Nexys2: Reset activo a nivel alto
      contseg <= 0;
      led0aux <= '0';
    elsif clk_50mhz'event and clk_50mhz='1' then
      contseg <= contseg + 1;
      if (contseg = 0) then
        led0aux <= not led0aux;
      end if;
    end if;
  end process;
end BEHAVIORAL;
```

*Código 4.2: Código de la primera versión (**con fallos**) del segundero para la Nexys2*

Una vez creado el fichero VHDL comprobamos su sintaxis. Para ello seleccionamos el fichero en la ventana *Sources*, y en la ventana *Processes*, hacemos doble clic en *Synthesize→Check Syntax*.

Si el resultado es correcto, ahora debemos indicar las restricciones temporales que tenemos. Entonces hacemos doble clic dentro de la ventana *Processes*, en *User Constraints→Create Timing Constraints*. Nos podrá salir un aviso de que va a crear un fichero `.ucf`, le damos a aceptar, pues en estos ficheros se guardan las restricciones y las asignaciones de los pines. Aparecerá la herramienta *Xilinx Constraints Editor* (figura 4.3) en ella seleccionamos la pestaña *Global* y en el reloj le indicamos su periodo: 10 (en nanosegundos) para el caso de la *XUPV2P* y 20 ns para *Nexys2*. Guardamos y cerramos, y volvemos a la ventana del *ISE*.

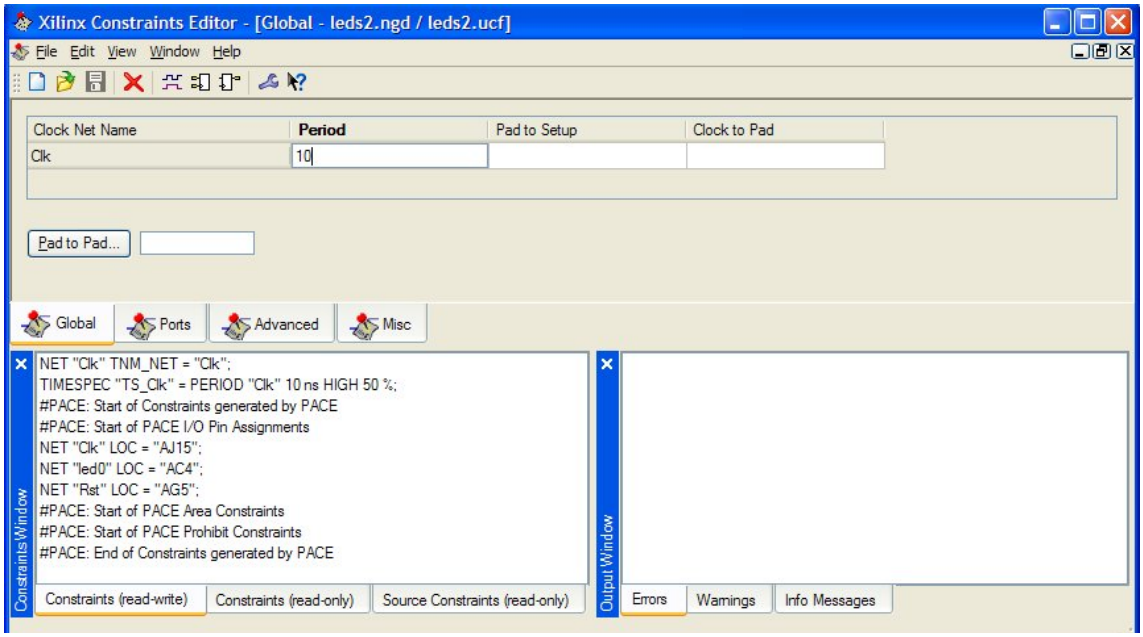


Figura 4.3: Xilinx Constraints Editor

En cuanto a las restricciones temporales, es importante señalar que las restricciones del reloj, son precisamente restricciones de nuestra placa, y le estamos indicando a la herramienta a qué frecuencia va nuestro reloj. Con este dato, la herramienta intentará ajustar la implementación del diseño a esas restricciones, y en caso de que no sea posible, nosotros debemos o bien usar un reloj con menor frecuencia (mediante el uso de componentes de la FPGA que generan relojes más lentos) u optimizar nuestro diseño para que sea más rápido.

Ahora asignaremos los pines, nos aseguramos que el módulo `seg1` está seleccionado en la subventana *Sources*, y en la subventana *Processes*, pinchamos en *User Constraints→Assing Package Pins*. Asignamos los pines de manera similar a la práctica anterior, en este caso para la *XUPV2P*: `clk_100mhz→AJ15 ; rst_n→AG5 ; led0→AC4`. Y para la *Nexys2*: `clk_50mhz→B8 ; rst→B18 ; led0→J14`.

Guardamos y cerramos, puede pasar que la herramienta pregunte por el formato del fichero `.ucf`, puede ser cualquiera, así que ponemos el primero de ellos.

Ahora, en la subventana *Processes* hacemos doble clic en el apartado *Implement Design*. Vemos si hay algún error o advertencia. Posteriormente, en la subventana central seleccionamos la primera pestaña *Design Summary*, y en ella localizamos *Performance Summary*, donde debe indicar si se han cumplido las restricciones temporales. Debe aparecer *All Contraints Met* y se puede pinchar en ella para analizar el informe con mayor detalle. Para este ejemplo tan sencillo, las restricciones temporales no iban a suponer ningún problema.

Ahora se puede generar el fichero de programación (.bit) y cargarlo en la FPGA mediante el *iMPACT* o el *Adept*, según la placa que tengamos.

Como resultado deberíamos ver parpadear el `LED0` cada segundo, si pulsamos el botón central se reinicia el parpadeo y si lo mantenemos pulsado, en la *XUPV2P* el led se queda encendido, ya que se le asigna un cero, que hace que se ilumine el led. En la *Nexys2* ocurre lo contrario al pulsar el reset.

4.2. Segundero (segunda versión)

¿Has descubierto el fallo del anterior circuito? ¿Te has fijado si el led permanecía encendido exactamente un segundo?

El circuito anterior tenía algún fallo. Cuando se crea un circuito para síntesis y se declara una señal entera con un rango se está especificando el ancho que tendrá dicha señal, es decir el número de bits. Y por tanto, sintetizar un diseño que tiene una señal con un rango que no es potencia de 2 puede crearnos problemas.

Por ejemplo, supongamos que necesitamos una señal con un rango de 0 a 9. Para representar esta señal en binario necesitamos cuatro bits. Si la declaramos según el código 4.3, independientemente de que hayamos indicado que su rango llega hasta 9, al sintetizar la señal va a tener 4 bits, y por tanto, su rango real va a ser de 0 a 15. Y si aún así quisiésemos limitar el rango, tendríamos nosotros que generar el hardware adecuado (con comparadores y otros componentes) para que nunca se pase de 9.

Por tanto, la manera adecuada de declarar la señal sería la mostrada en el código 4.4. Siendo la declaración de `dec2` la más adecuada, pues con ella, al usar la operación

potencia (**) estamos especificando el número de bits. No debemos olvidar restar uno, pues si no, estaríamos usando un bit más.

```
signal dec0 : natural range 0 to 9;
```

Código 4.3: declaración de un rango "incorrecta" para síntesis

```
signal dec1 : natural range 0 to 15;
signal dec2 : natural range 0 to 2**4-1;
```

Código 4.4: declaraciones correcta de un rango

Es importante tener en cuenta que si declaramos una señal entera o natural sin rango (código 4.5), el sintetizador implementará la señal con 32 bits (32 cables). Si no necesitamos tantos bits para la señal estaremos desperdiciando recursos de la FPGA.

```
signal sin_rango : natural;   -- por no tener rango se sintetiza con 32 bits
```

Código 4.5:la declaración de una señal sin rango implementa una señal de 32 bits

Una opción recomendable en el uso de señales, es el empleo del tipo `unsigned` en vez de `natural`, y el uso de `signed` en vez de `integer`. El uso de estos tipos se explicó en el capítulo 6 de la referencia [17mach], también se verá con más detalle en el capítulo 8. Un ejemplo de declaración de señal `unsigned` se muestra en el código 4.6.

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

Código 4.6: Declaración de señal `unsigned`

Es probable que con esta información ya sepas por qué no funciona correctamente la versión anterior del segundero.

En el código 4.1 creamos la señal `contseg` con un rango de 100 millones (ó 50 millones para la Nexys2, código 4.2). Esto lo hicimos porque teníamos dividir la frecuencia entre 100 millones (ó 50 millones) para pasar de 100 MHz (ó 50 MHz) a 1 Hz. Sin embargo, como acabamos de ver, indicar que una señal tiene un rango de 100 millones no nos asegura que físicamente esa señal tenga ese rango. De hecho, el rango de esa señal será la potencia de 2 inmediatamente superior, en este caso, algo más de 134 millones (2^{27}). Para el caso de la Nexys2 el rango sería algo más de 67 millones (2^{26}). Por lo tanto, nuestro led va a estar encendido algo más de 1,34 segundos en vez de estar un segundo luciendo.

Para comprobarlo, creamos un nuevo proyecto `SEG2` (no cambies el proyecto anterior porque lo vamos a usar), que será una ampliación del diseño de la anterior práctica. En este proyecto vamos a contar segundos de tres maneras diferentes, por tanto, en vez de un led usaremos tres para mostrar las tres cuentas. La entidad para la *XUPV2P* se muestra en el código 4.7 y para la *Nexys2* se muestra en el código 4.8:

```
entity SEG2_XUP is
  Port (
    rst_n        : in   std_logic;
    clk_100mhz   : in   std_logic;
    led0         : out  std_logic;
    led1         : out  std_logic;
    led2         : out  std_logic
  );
end SEG2_XUP;
```

Código 4.7: Entidad de la segunda versión del segundero para la XUPV2P

```
entity SEG2_NEXYS is
  Port (
    rst        : in   std_logic;
    clk_50mhz  : in   std_logic;
    led0       : out  std_logic;
    led1       : out  std_logic;
    led2       : out  std_logic
  );
end SEG2_NEXYS;
```

Código 4.8: Entidad de la segunda versión del segundero para la Nexys2

Por tanto, en la arquitectura usaremos tres contadores, cuyas señales declararemos así:

```
architecture Behavioral of SEG2_XUP is
  signal cont0 : natural range 0 to 100000000;
  signal cont1 : natural range 0 to (2**27)-1;
  signal cont2 : natural range 0 to (2**27)-1;
....
```

Código 4.9: Declaración de las señales usadas para contar

La señal `cont0` tiene un rango de 10^8, mientras que la señal `cont1` y `cont2` tienen un rango mayor que se corresponde con la potencia de 2 inmediata superior a 10^8.

Vamos a implementar los contadores van a contar de la siguiente manera:

- `cont0` va a hacer parpadear a `led0` de la misma manera que en la práctica anterior.
- `cont1` va a hacer parpadear a `led1` de la misma manera que en la práctica anterior, la diferencia con `cont1` es el rango. El rango de `cont0` es menor que el de `cont1`.
- `cont2` va a hacer parpadear a `led2` pero no se va a desbordar, sino que controlaremos que cuando llegue a 10^8-1 se inicialice. En caso de la Nexys2, cuando el contador llegue a 50 millones.

Una vez que tenemos el diseño y hemos comprobado su sintaxis, lo implementamos en la FPGA, y vemos si `cont0` parpadea a la frecuencia de `cont1` o de `cont2`. Si parpadea a la frecuencia de `cont1` significa que ha superado el rango que le habíamos impuesto en la declaración y que imponer rangos que no sean potencias de dos nos puede llevar a errores.

5. Simulación con Modelsim

En este capítulo vamos a ver cómo simular con Modelsim[20]. En el manual de prácticas de ED2 [17mach] se explicó cómo simular con el simulador del ISE (capítulo 8). Es conveniente que repases ese capítulo, pues describe los fundamentos de los bancos de pruebas.

Hasta ahora hemos realizado circuitos simples que en el caso de que hayamos tenido algún error, no era muy difícil de corregir. Sin embargo casi nunca es así, y conviene simular el circuito para comprobar si hay algún error antes de implementarlo en la FPGA. De hecho, lo habitual es que los diseños no funcionen a la primera, y por tanto, la simulación se convierte un paso imprescindible.

Vamos a crear un banco de pruebas (en inglés *test bench*) dentro del proyecto anterior (apartado 4.1). El banco de pruebas es un diseño VHDL que sirve para comprobar el funcionamiento de un circuito. Este diseño VHDL no tiene que estar limitado a las estructuras para síntesis, sino que puede usar todo el conjunto del VHDL. Esto se debe a que el banco de pruebas no va a implementarse en la FPGA (por lo tanto no se va a sintetizar) sino que se utilizará únicamente para simulación.

Así que creamos una nueva fuente de tipo *VHDL TestBench* y la llamaremos `tb_seg1.vhd` (figura 5.1).

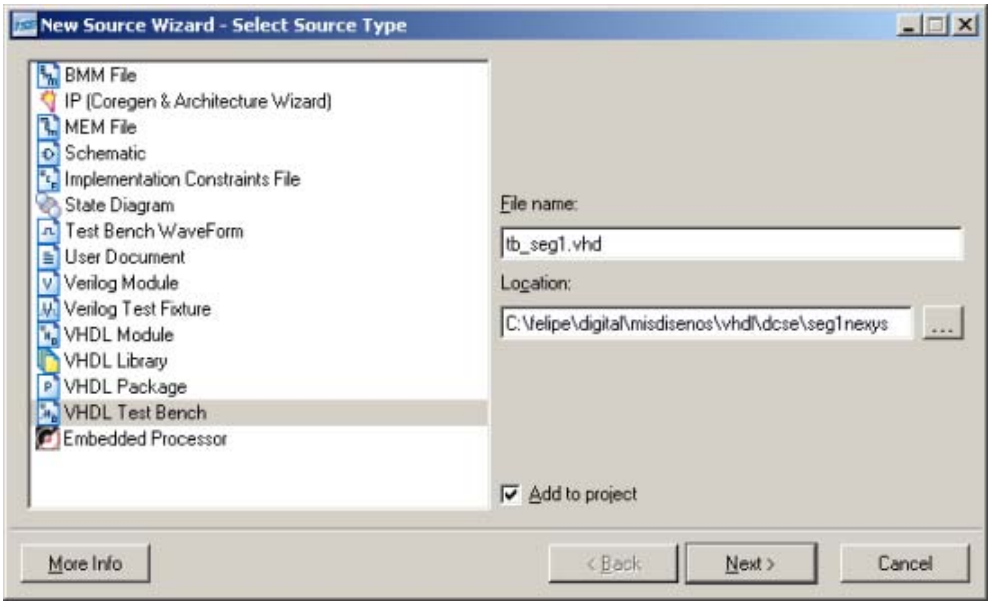


Figura 5.1: Creación de un nuevo banco de pruebas

Pinchamos en *Next* o *Finish* en todas las ventanas que nos aparezcan hasta llegar a la edición del fichero del banco de pruebas. El ISE nos ha creado la estructura del banco de pruebas (figura 5.2) y ahora tenemos que crear los procesos que generen los estímulos. Si tienes dudas sobre la estructura del banco de pruebas de la figura 5.2, te recomendamos

[20] Modelsim (http://model.com) es un simulador de la compañía Mentor Graphics. Tienen una versión limitada para estudiantes y también ofrece una versión para Xilinx: Modelsim XE, disponible desde la página web de Xilinx.

que repases el capítulo 8 del manual de prácticas de ED2 [17mach], donde se explica con detalle cada una las partes del banco de pruebas.

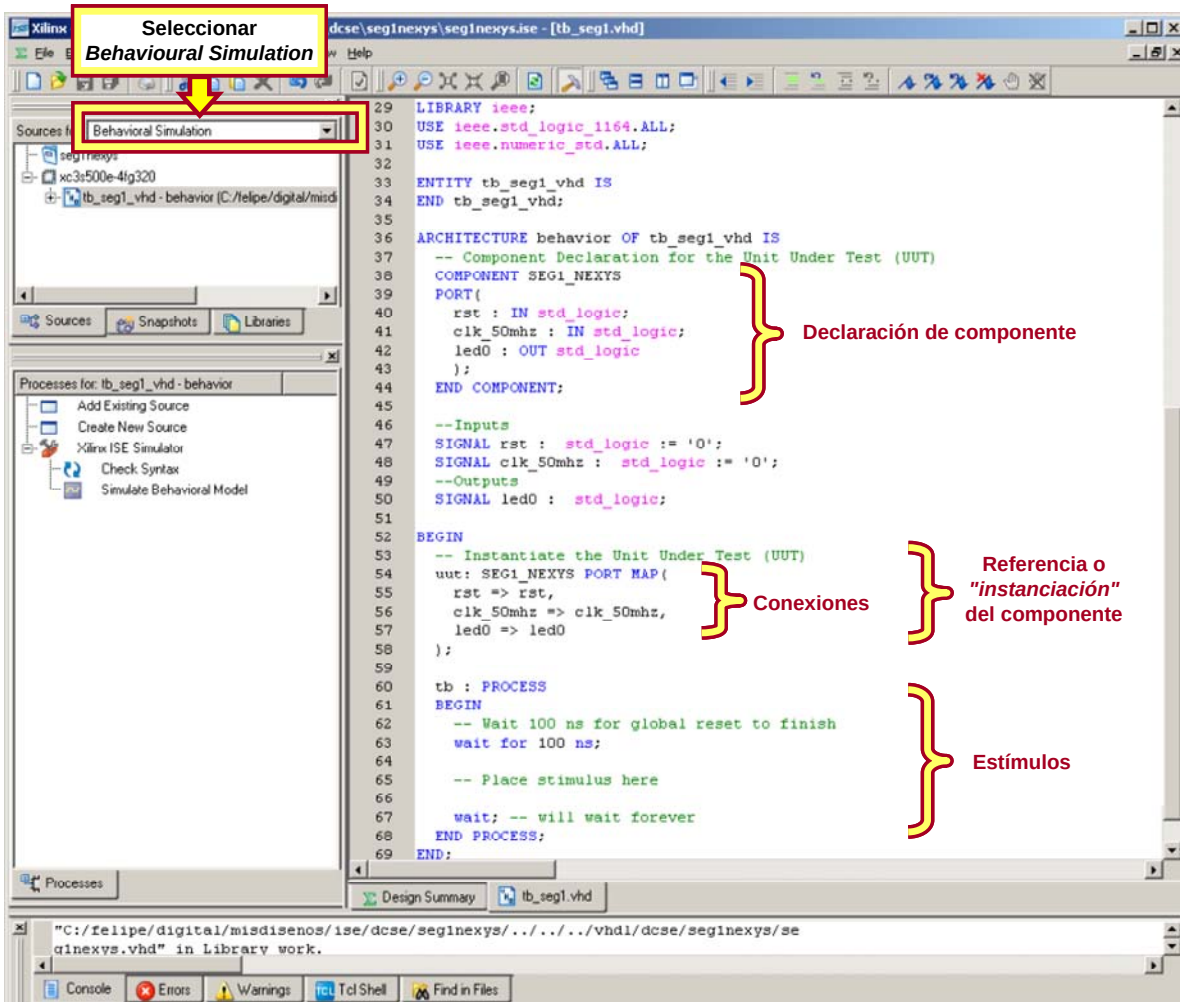


Figura 5.2: Estructura del banco de pruebas

Dentro de la zona señalada como estímulos de la figura 5.2 hay una plantilla de un proceso creada por el *ISE*. Nosotros vamos a crear dos procesos. Uno para la señal que va a simular la señal de reloj y otro para la señal de reset. No hace falta más, ya que este circuito sólo tenía estas dos señales de entrada. Para crear los procesos, o bien borras la plantilla creada y añades los procesos nuevos, o bien modificas la plantilla para que el proceso quede como se muestra a continuación.

El proceso que simula la señal de reloj se muestra en los códigos 5.1 y 5.2. La diferencia entre ambos es la frecuencia del reloj. El código 5.1 genera un reloj de 10 ns (5 + 5 ns), que es el reloj de la *XUPV2P* (100 MHz). Por otro lado, el código 5.2 genera un reloj de 20 ns (10 + 10 ns), que es el reloj de la *Nexys2* (50 MHz).

```
PClk_XUP: Process
begin
  clk_100mhz <= '0';
  wait for 5 ns;
  clk_100mhz <= '1';
  wait for 5 ns;
end process;
```

Código 5.1: Proceso que simula el reloj de la XUPV2P

```
PClk_Nexys: Process
begin
  clk_50mhz <= '0';
  wait for 10 ns;
  clk_50mhz <= '1';
  wait for 10 ns;
end process;
```

Código 5.2: Proceso que simula el reloj de la Nexys2

Observa que estos procesos no tienen lista de sensibilidad. Esto es porque los procesos que tienen sentencias con `wait` no tienen lista de sensibilidad. Estos procesos no son válidos para síntesis[21], ya que al hardware no le puedes decir directamente que espere 5 nanosegundos. Para ello, tendrías que realizar un contador que contase los ciclos de un reloj cuya frecuencia sea conocida (de manera similar a como se hizo en la práctica anterior).

Durante la simulación, los procesos de los códigos 5.1 y 5.2 se ejecutan permanentemente. Se detiene durante 5 ns (ó 10 ns) cada vez que llega a una de las sentencias `wait`. Por tanto, la señal de reloj (`clk_100mhz` ó `clk_50mhz`) recibe periódicamente un valor diferente..

Para el proceso de la señal de reset nos podría bastar con simular que pulsamos el botón de reset una vez. Los códigos 5.3 y 5.4 muestran estos procesos para la *XUPV2P* y la *Nexys2*. La única diferencia de estos dos procesos es cómo se activan los reset: en la *XUPV2P* a nivel bajo y en la *Nexys2* a nivel alto.

```
PRst_N_XUP: Process
begin
  rst_n <= '1';
  wait for 108 ns;
  rst_n <= '0';   -- Reset a nivel bajo
  wait for 75 ns;
  rst_n <= '1';
  wait;   -- esperamos "eternamente"
          -- con el reset desactivado
end process;
```

Código 5.3: Proceso que simula la señal de reset para la XUPV2P

```
PRst_Nexys: Process
begin
  rst <= '0';
  wait for 108 ns;
  rst <= '1';   -- Reset a nivel alto
  wait for 75 ns;
  rst <= '0';
  wait;    -- esperamos "eternamente"
           -- con el reset desactivado
end process;
```

Código 5.4: Proceso que simula la señal de reset para la Nexys2

En estos ejemplos, el reset se ha activado en un tiempo "aleatorio" de 108 ns. Podía haberse elegido otro. Y se ha mantenido activo durante 75 ns. En realidad, estos tiempos tendrían que ser mucho mayores si se quisiese representar el tiempo que aproximadamente se mantienen presionando los pulsadores, pero es innecesario alargar la simulación durante el reset.

La última sentencia de estos procesos es un `wait` que termina en punto y coma, sin tiempos de espera. Este `wait` hace que se detenga la ejecución del proceso y se quede esperando "eternamente". Por lo tanto, a partir de entonces el reset se quedará inactivo durante toda la simulación.

Para simular el circuito, podemos realizarla de dos maneras (lo haremos de la segunda forma):

- Para la primera forma, necesitamos haber seleccionado el *Modelsim* como el simulador dentro del ISE (recuerda la figura 3.8.). Seleccionamos *Behavioral Simulation* en la subventana

[21] Procesos con sentencia "`wait for`", un proceso con sentencia "`wait until`" puede ser sintetizable aunque no los veremos.

Sources (pestaña *Sources*), ver figura 5.2. Para simular el modelo podemos pinchar dentro de la ventana *Processes*, en *Modelsim Simulator*

- La segunda forma es crear un proyecto en *Modelsim*. Esta manera se explicará a continuación.

Creamos una carpeta llamada `simulacion` en la carpeta donde tengamos nuestro proyecto. Abrimos el *Modelsim* y creamos un nuevo proyecto: *File→New→Project*. Y creamos el proyecto *seg1* dentro de la carpeta *simulacion*.

Añadimos al proyecto los ficheros VHDL que hemos creado desde el *ISE*: *File→Add to Proyect→Existing File*.

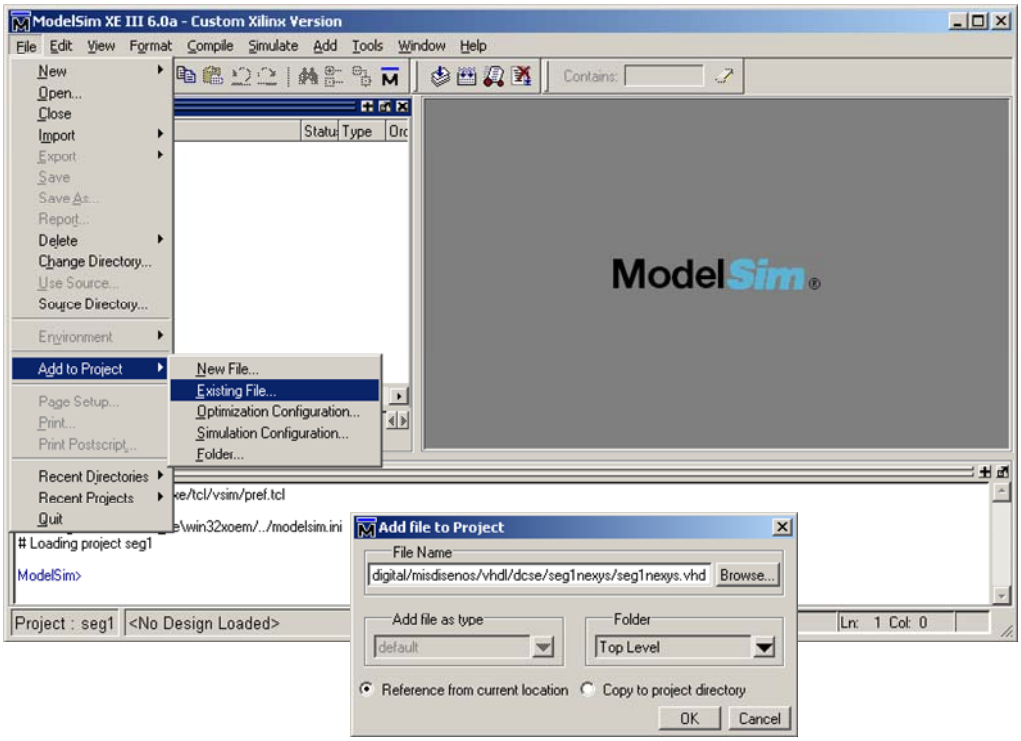


Figura 5.3: Añadiendo ficheros al proyecto de Modelsim

Una vez que aparecen los dos ficheros en la ventana de la izquierda, seleccionamos ambos y con el botón derecho del ratón le damos a *Compile→Compile All*, y vemos si tenemos algún error. En caso de que haya algún error de compilación se mostrará en la ventana inferior en letras rojas, y pinchando dos veces en el mensaje nos saldrá una ventana con la información del error y el número de línea. También se puede pinchar en el mensaje para que se muestre el error en el código. Si hubiese un error, lo corregimos, grabamos y volvemos a repetir el proceso. En la ventana izquierda deberán aparecer ambos ficheros con una marca verde que indica que se han compilado con éxito.

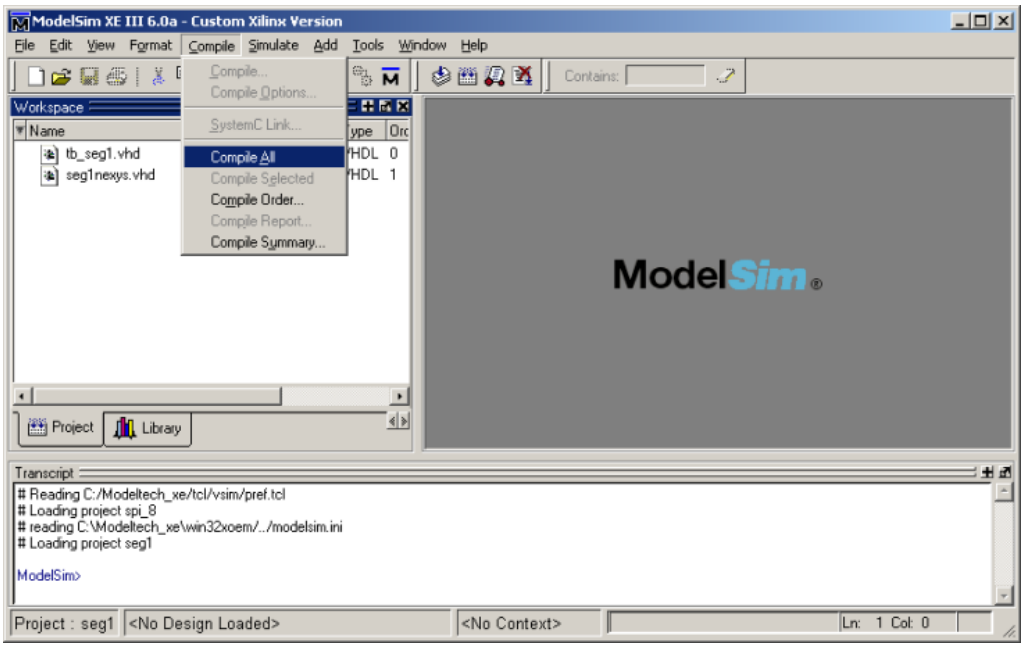


Figura5.4: Compilando los ficheros del proyecto de Modelsim

Para empezar la simulación, pinchamos en *Simulate→Start Simulation*. Nos saldrá una ventana, en ella entramos en *work*, y dentro de ella seleccionamos el banco de pruebas (`tb_seg1_vhd`), y le damos a *OK*.

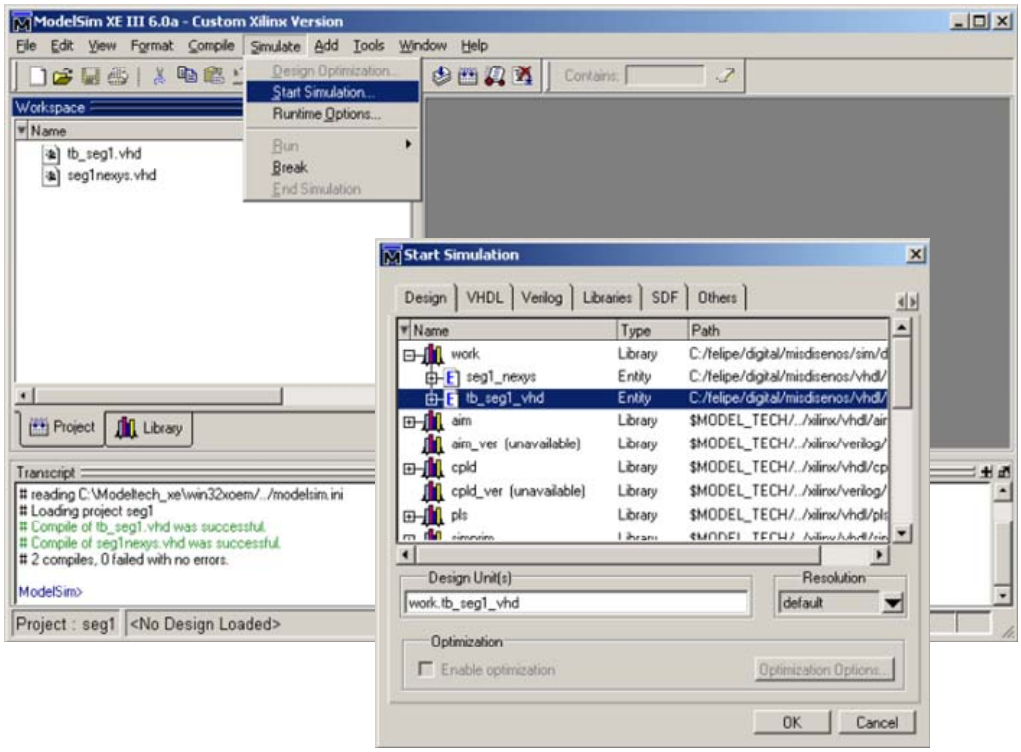


Figura 5.5: Comienzo de la simulación y selección del fichero del banco de pruebas para simular

Según cómo tengamos configurado el *Modelsim*, saldrán unas ventanas u otras. Nosotros queremos que salgan las ventanas *Objetc* y *Wave*. Para ello, si no estuviesen, pinchamos en *View→Debug Windows→Wave* y en *View→Debug Windows→Objects*.

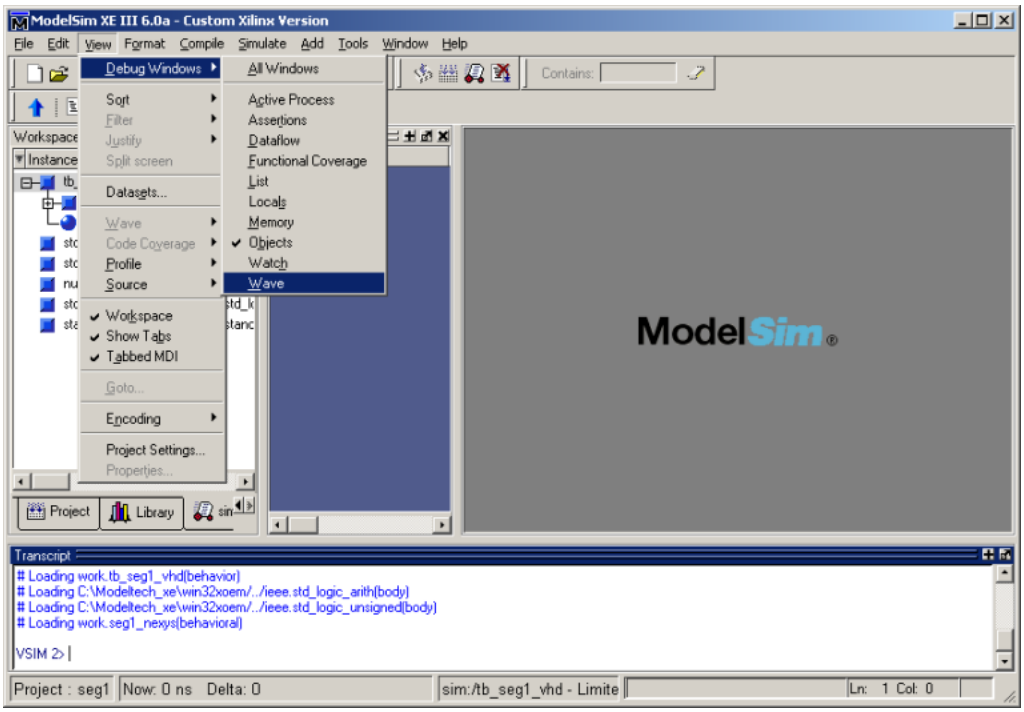


*Figura 5.6: Añadiendo la ventana de las formas de onda (*Wave*)*

Para incluir las señales en la ventana de simulación (*Wave*) y así ver las formas de onda, nos ponemos en la ventana de *Workspace* y seleccionamos el componente UUT. Haciendo esto, en la ventana *Objects* aparecerán las señales de ese componente y pinchando con el botón derecho del ratón dentro de la ventana *Objects* seleccionamos *Add to Wave→Signals in Region*. Si quisiésemos añadir todas las señales de circuitos, haríamos *Add to Wave→ Signals in Design*. O bien podemos añadir las señales individualmente.

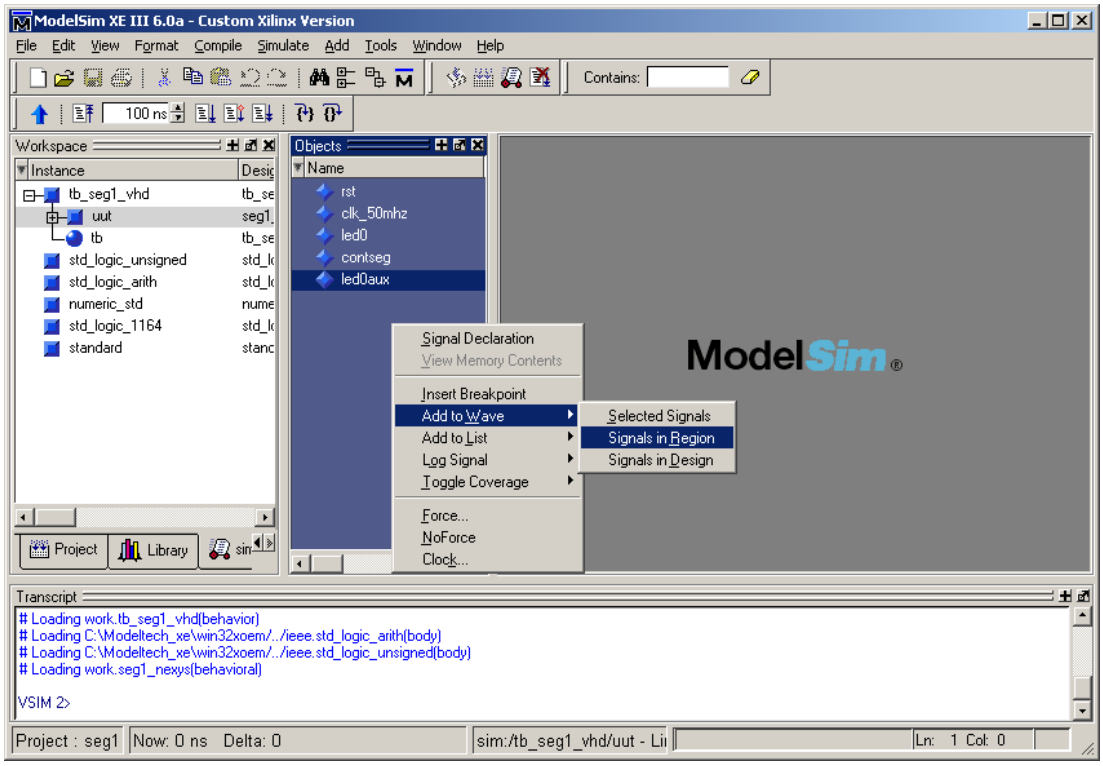


Figura 5.7: Añadiendo las señales de un componente

Ahora queremos simular el circuito durante 1400 ms (un poco más de un segundo). Esto lo indicamos en la ventanita para poner el tiempo de simulación (ver figura 5.8). Y pinchamos en el botón de la derecha (*Run*). Seleccionamos la pestaña *wave* si no estuviese seleccionada.

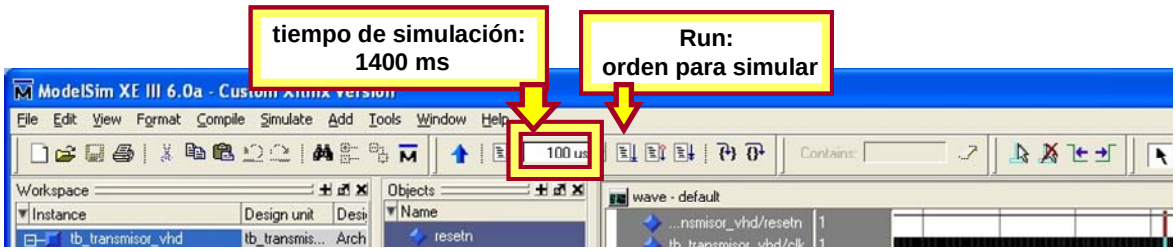


Figura 5.8: Indicación del tiempo de simulación y orden para empezar la simulación

Es una simulación larga, pues un segundo de simulación son muchos ciclos de reloj. Al sobrepasar el segundo de simulación se producirá un error por haberse desbordado el contador. En este caso, la simulación nos advierte del error que vimos en el apartado anterior: hemos definido la señal con un rango que se va a desbordar (*out of range*) y no hemos limitado la cuenta (ver figura 5.9).

Para ver las formas de onda, en caso de que no se muestre, pincha en la pestaña de las formas de onda. Para ver la simulación completa, pincha en el icono mostrado en la figura 5.9. Al lado del icono tienes otros para ampliar o disminuir el detalle (zoom). Puedes extraer la ventana de las formas de onda y hacerla independiente de la ventana del Modelsim.

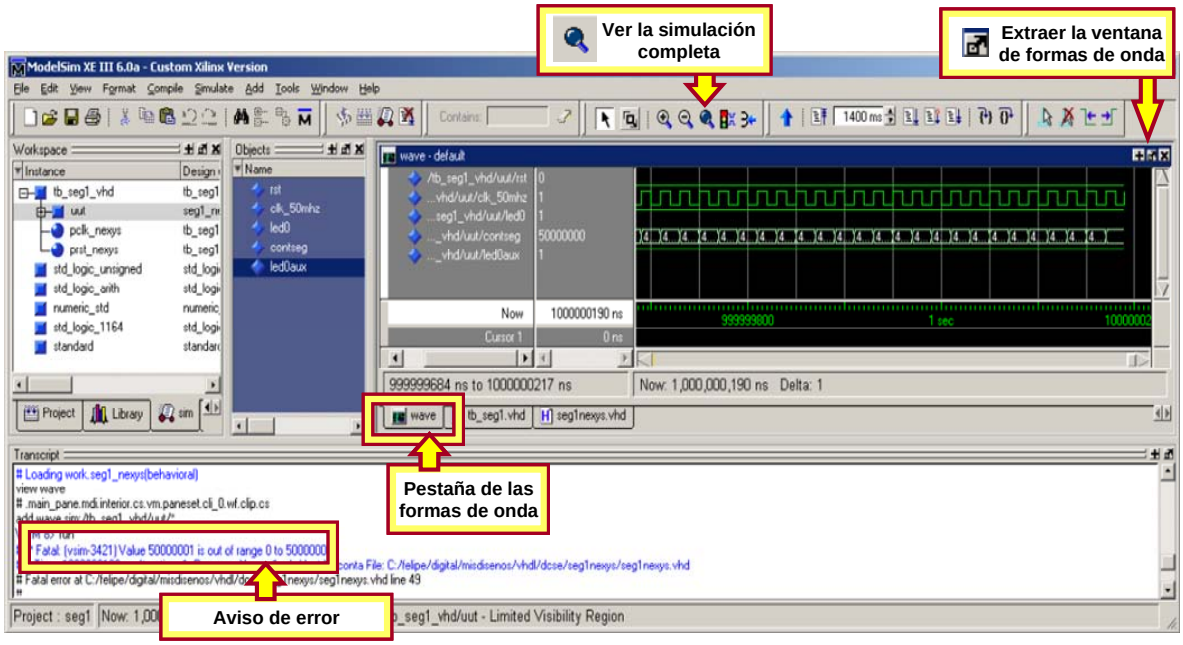


Figura 5.9: Resultado de la simulación

En capítulos posteriores iremos profundizando en los bancos de pruebas y la simulación.

6. Constantes, genéricos y paquetes

En este capítulo repasaremos algunos conceptos y técnicas que nos facilitarán el diseño en VHDL, especialmente si necesitamos adaptar nuestro diseño a otras FPGA, tarjetas o entornos. Esto es muy importante porque hará que nuestro diseño sea más fácilmente reutilizable, lo que nos permitirá ahorrar tiempo de desarrollo.

6.1. Uso de constantes

De las prácticas anteriores podemos haber notado la incomodidad que implica utilizar dos placas con algunas características diferentes. Por ejemplo, en la placa *XUPV2P* para encender un led necesitamos poner un '0', mientras que en la *Nexys2* tenemos que poner un '1' para encenderlo. También vimos que las placas relojes con distinta frecuencia.

Por último, si hemos utilizado la *XUPV2P*, quizá habremos notado que utilizar lógica negada puede dificultarnos la implementación y la comprensión del funcionamiento un circuito. Por ejemplo, si queremos encender el led `ld0` cuando presionamos el pulsador `pb0` **y** tengamos el interruptor `sw0` conectado, por ser lógica negada, la sentencia no es muy intuitiva, pues no se forma con la `and` (código 6.1), sino con la `or` (código 6.2). Esto, aunque es fácilmente explicable con las leyes de De Morgan, nos puede provocar algo de confusión.

```
ld0 <= pb0 and sw0;
```

Código 6.1: Sentencia con lógica directa

```
ld0 <= pb0 or sw0;
```

Código 6.2: Sentencia equivalente al código 6.1 pero con lógica inversa

Para evitar la confusión que nos puede dar lugar al leer el código 6.2 tenemos varias opciones:

1. Usar nombres que indiquen que son señales con lógica negada. Esto nos advierte de la situación, sin embargo tenemos que seguir trabajando con lógica negada. El código 6.3 nos muestra un ejemplo de cómo podríamos indicar que las señales funcionan con lógica negada (añadiendo `"_n"`).
2. Usar lógica directa internamente, habiendo negado los puertos de entrada antes de usarlos y negando después las salidas. A los nombres de los puertos les podemos aplicar la recomendación anterior. Como ejemplo se muestra el código 6.4, en el que las tres primeras sentencias transforman la lógica de los puertos. A raíz de esto, la última sentencia ya trabaja con lógica directa.

```
  ld0_n <= pb0_n or sw0_n;
```

Código 6.3: Añadiendo '_n' a las señales que usan lógica inversa

```
  -- asignacion de los puertos
  -- pb0_n, sw0_n : puertos entrada
  pb0   <= not pb0_n;
  sw0   <= not sw0_n;
  -- ld0_n, : puerto de salida
  ld0_n <= not ld0;

  -- Ya trabajamos con logica directa:
  ld0 <= pb0 and sw0;
```

Código 6.4: Invirtiendo los puertos que trabajan con lógica inversa

3. Usar constantes que nos digan cuál es el valor de funcionamiento de las señales (código 6.5). Con esto hacemos el circuito independiente del tipo de lógica, lo único que tenemos que hacer es poner el valor adecuado de las constantes. Esta opción tiene la gran ventaja de que nos permite usar el mismo diseño para la *XUPV2P* y la *Nexys2* con solo cambiar el valor de las constantes. El inconveniente es que la sentencia es más larga.

```
architecture ....
  constant c_on  : std_logic := '0'; --solo tenemos que cambiar aqui según la lógica
  constant c_off : std_logic := not c_on;
begin
   ld0 <= c_on when ((pb0 = c_on) and (sw0 = c_on)) else c_off;
end ...;
```

Código 6.5: Usando constantes para independizar el circuito del tipo de lógica de las entradas

Siguiendo esta última indicación, puedes probar a cambiar el código 3.5 de modo que funcione igual que lo haría en la *Nexys2* (código 3.6).

6.2. Uso de genéricos

Si tenemos un diseño grande, con muchos ficheros (recuerda el VHDL estructural del apartado 2.10), puede ser incómodo y provocar errores el tener que cambiar el valor de las constantes en cada uno de los ficheros, ya que en cada uno de los ficheros tendríamos declaradas las mismas constantes (como en el código 6.5).

El VHDL nos permite definir un tipo de constantes especiales llamadas genéricos (*generic*). Estas constantes se declaran en las entidades y sus valores se pueden propagar al referenciar los componentes.

En el código 6.6 se muestra la adaptación del código 6.5 asignando el genérico al valor de la constante.

```
entity gen_xup is
  -- los genericos van en la entidad antes de los puertos
  Generic (
    g_logic : std_logic := '0'
  );
  Port (
    pb0 : in   std_logic;
    sw0 : in   std_logic;
    ld0 : out  std_logic
  );
end gen_xup;

architecture Behavioral of gen_xup is
  constant c_on  : std_logic := g_logic;  -- el generico se puede asignar a la cte
  constant c_off : std_logic := not c_on;
begin
  ...
```

Código 6.6: Declaración y uso de genéricos

Como se ha comentado, la ventaja del genérico es que se puede propagar por el VHDL estructural. El paso de los valores se hace de manera similar a los puertos, en vez de la

sentencia `PORT MAP` se utiliza `GENERIC MAP`. Al propagar los genéricos se pierde el valor que se le había dado en la declaración de la entidad. De hecho, por esto mismo no es necesario asignar un valor en la declaración.

Aunque más adelante veremos ejemplos de su utilización en circuitos estructurales, en el código 6.7 se muestra un ejemplo.

```
GEN: GEN_XUP
  generic map (
    g_logic     => g_tipolog, -- el generico g_logic recibira el valor de g_tipolog
    g_generico2 => g_generico2
  )
  port map (
    pb0      => pb0,
    sw0      => clk,
    ld0      => ld0
  );
```

Código 6.7: Ejemplo de la transmisión de los genéricos en una referencia a componente

Aunque el uso de genéricos tiene muchas ventajas, no siempre es conveniente usarlos. Algunas herramientas sólo admiten algunos tipos concreto de datos para los genéricos. Y por otro lado, no conviene declarar una gran cantidad de genéricos. Algunos recomiendan el uso de genéricos sólo para ciertos aspectos en concreto, por ejemplo para simulación (definir retardos o condiciones variables), o también cuando según determinadas condiciones se quiera variar el valor de los genéricos de un componente. Esta técnica, aunque muy útil, por ahora no la vamos a ver.

6.3. Uso de paquetes

Una alternativa al uso de genéricos es declarar las constantes en un paquete. El VHDL permite declarar constantes, tipos, funciones, procedimientos, etc. en paquetes de modo que se puedan utilizar en todo el diseño sin tener que declararlos en cada fichero.

De hecho, ya hemos estado usando paquetes al incluir las siguientes sentencias (código 6.8) en cada fichero VHDL.

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

Código 6.8: Inclusión del paquete `STD_LOGIC_1164` de la biblioteca IEEE

Con la primera sen estamos diciendo que vamos a incluir la biblioteca IEEE. Con la segunda sentencia decimos que de la biblioteca IEEE vamos a usar el paquete `STD_LOGIC_1164`. El `ALL` del final indica que podremos usar todo lo que incluye el paquete.

Ahora crearemos un sencillo proyecto en el que incluiremos un paquete. El nombre del proyecto será `led_pkg`.

Un paquete se crea en un fichero aparte. Para crear un nuevo paquete, desde la herramienta ISE crearemos una nueva fuente como ya sabemos hacer, pero indicaremos que es de tipo *VHDL Package*. Al hacerlo la herramienta ISE incluye unos patrones que nos pueden servir de ayuda para recordar la sintaxis. De todos modos, como nuestro paquete va a ser muy sencillo nos basta con copiar el código 6.9.

```
package CTE_PKG is
  --  Paquete de definicion del nivel logico de las constantes
   constant c_on  : std_logic := '0'; --solo tenemos que cambiar aqui según la lógica
   constant c_off : std_logic := not c_on;
end CTE_PKG;
```

Código 6.9: Paquete con la definición de las constantes

En el código 6.9, la constante de encendido `c_on` se ha puesto a `'0'`, por tanto correspondería a la versión de la *XUPV2P*, para la *Nexys2* bastaría con poner esta constante a `'1'`.

En la herramienta ISE los paquetes no están a la vista desde la ventana *Sources*. Para poder editar un paquete tienes que pinchar en la pestaña *Libraries* y desplegar la carpeta *work*, allí verás si el paquete está incluido o no. Pinchando en él podrás editarlo. En la figura 6.1 se muestra una ventana que muestra la anterior explicación.

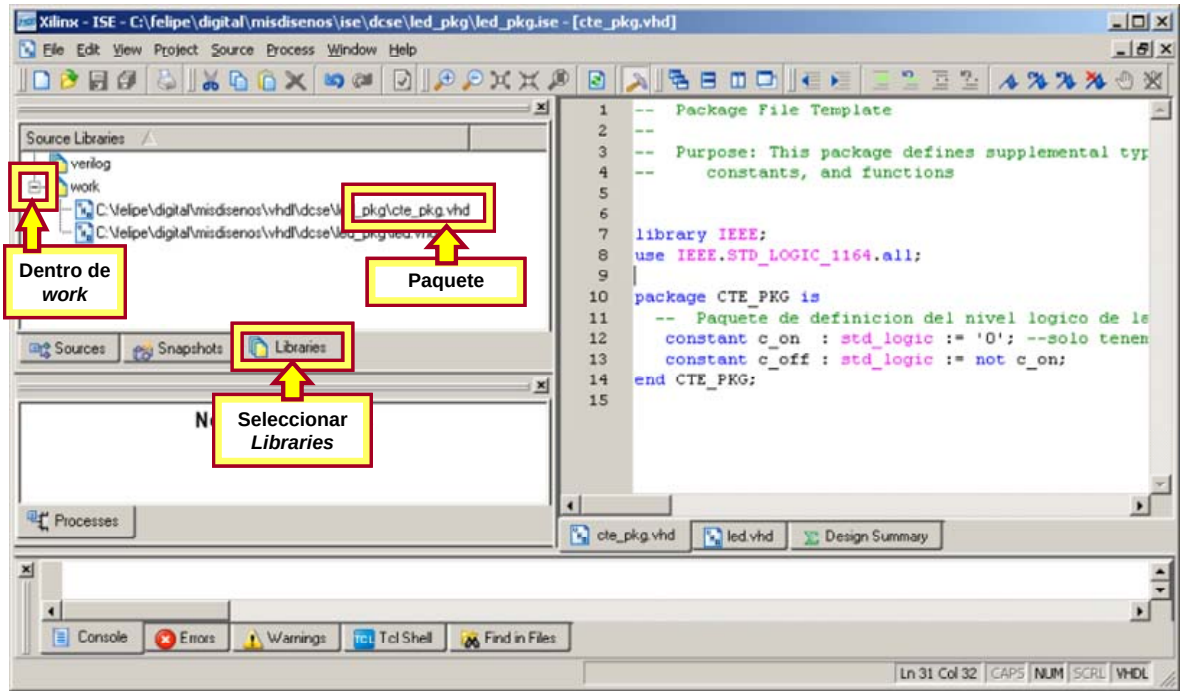


Figura 6.1: Cómo ver el paquete incluido en el proyecto

Ahora crearemos una entidad que va a usar este paquete. La entidad simplemente tendrá que apagar el led `ld0` y encender el `ld1`. La entidad sería como la mostrada en el código 6.10.

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

library WORK;              -- la biblioteca por defecto del diseno se llama WORK
use WORK.CTE_PKG.ALL;

entity led is
  Port ( ld0 : out  STD_LOGIC;
         ld1 : out  STD_LOGIC);
end led;

architecture Behavioral of led is
begin
  ld0 <= c_off;
  ld1 <= c_on;
end Behavioral;
```

Código 6.10: Entidad y arquitectura que usan el paquete del código 6.9

Observando el código 6.10 podemos apreciar que no tenemos que modificar la entidad y la arquitectura para usarlas en la *XUPV2P* o en la *Nexys2*, nos bastaría con modificar el valor de la constante del paquete (código 6.9). Esto nos permite realizar diseños más fácilmente adaptables a otros entornos. En nuestro caso, podemos usar ambas placas con menores inconvenientes.

En el código 6.10 podemos observar que la biblioteca del paquete que hemos creado se llama `WORK`. Esta es la biblioteca por defecto, la biblioteca de trabajo y por lo general, pondremos nuestros paquetes en esa biblioteca (si no decimos nada, estarán ahí).

Ahora, prueba a implementar este diseño en la *XUPV2P* y en la *Nexys2*. Lo único que tienes que hacer es crear el paquete del código 6.9 y la entidad y arquitectura del código 6.10. La constante del paquete la tendrás que cambiar según en qué placa lo estés implementando, igual que el fichero `.ucf`, que éste inevitablemente va a ser distinto, pues es específico para cada placa.

7. Transmisión en serie por RS-232

El objetivo de esta práctica es realizar un diseño de mediana complejidad. Para la realización de este circuito es importante que tengas claro la teoría de los registros de desplazamiento y las máquinas de estados finitos (FSM[22]) y cómo se implementan en VHDL. Si tienes dudas o te has olvidado, te recomendamos que realices las prácticas de ED2, en concreto los capítulos 7, 9 y posteriores del libro [17mach]. Una buena práctica sería que las implementases en las placas *XUPV2P* y *Nexys2*.

En esta práctica vamos a establecer una comunicación asíncrona entre la placa y una computadora. Es un buen ejemplo de **diseño jerárquico y configurable**, en el que además se van a repasar las máquinas de estados finitos y registros de desplazamiento, y nos servirá para aplicar conceptos vistos en prácticas anteriores, como los divisores de frecuencia y el uso de paquetes en VHDL.

Por otra parte, y aunque en el capítulo 5 de este manual ya se utilizó la herramienta de simulación Modelsim, en esta práctica se va a explicar en detalle cómo hacer un banco de pruebas (*testbench*) y como interpretar los resultados que se obtienen del simulador.

Debido a que esta práctica es larga y se aplican conceptos importantes, en el apartado siguiente se resume su contenido.

7.1. Desarrollo de la práctica

En primer lugar, se expondrá la metodología propuesta para afrontar el diseño de un circuito complejo, en la que es fundamental realizar una partición del circuito en módulos más sencillos.

Posteriormente, en el apartado 7.3 se explicará brevemente el funcionamiento de una UART y el envío de datos mediante el protocolo serie RS-232: tamaño de datos, trama de datos, señales que se utilizan en la comunicación, velocidad de transmisión, etc.

A continuación se propone un **diseño modular**, en el que primero se hará el transmisor (en el apartado 7.4), después el receptor para posteriormente integrarlos en un único diseño.

Para diseñar el **transmisor** lo primero que se hace es identificar las entradas y salidas. A continuación se plantea su diagrama de bloques. El diseño que se propone es un diseño **configurable**, en el que el uso de constantes, funciones y paquetes facilita la reutilización y reduce las fuentes de errores al utilizar dos placas distintas en las prácticas. Una vez diseñado el transmisor se procede a su comprobación (apartado 7.5). Para ello crearemos el banco de pruebas, y mediante un simulador (el del ISE o el ModelSim) verificaremos el funcionamiento del circuito.

Para terminar con el transmisor, lo implementaremos en la FPGA y nos comunicaremos con la computadora mediante el puerto serie.

De forma análoga a como se ha realizado el transmisor se diseñará el receptor, para posteriormente implementar la UART completa. En un diseño estructural conectamos el

[22] FSM: del inglés: *Finite State Machine*

transmisor y el receptor de manera que lo recibido por el receptor se emita por el transmisor, obteniéndose el eco en la computadora.

7.2. Consideraciones para el diseño de un circuito

Antes de explicar qué es una UART y abordar su diseño, vamos a introducir unas consideraciones previas, a tener en cuenta no sólo para esta práctica sino también cada vez que nos enfrentemos con el diseño de un circuito digital.

Como ya se ha comentado anteriormente el objetivo fundamental que se persigue con este manual es aprender a diseñar circuitos digitales de relativa complejidad usando VHDL y dispositivos lógicos programables (CPLD o FPGA). Para conseguir este objetivo, a lo largo del manual se propone un metodología para el diseño de estos circuitos. Para ello los pasos a seguir son:

- Estudio de las especificaciones
- División funcional
- Integración de cada uno de los bloques (o módulos).

La partición del circuito en bloques (división funcional) es muy importante para poder abordar un diseño complejo. En el apartado 2.10 vimos las ventajas de realizar la partición, algunas de ellas son:

- Independizar los bloques: cada una de las partes son independientes
- Poder repartir el trabajo entre varias personas
- Simplificar la complejidad del diseño, pues los bloques son más sencillos. Esto hace que sea más fácil de diseñar y de verificar.
- Simplificar la verificación del diseño. Como los bloques están verificados de manera independiente, la verificación del conjunto resulta más sencilla.
- Y otras ventajas como mayor facilidad para reutilizar los bloques, modificarlos, usar otros ya diseñados, ...

Para cada uno de los bloque funcionales en que se dividida el proyecto habrá que identificar entradas y salidas, realizar el diseño y por último, comprobar su correcto funcionamiento. Para verificar que nuestro módulo hace lo que tiene que hacer, y no hace lo que no tiene que hacer, se utilizarán las herramientas de simulación. Mediante la generación de estímulos y la interpretación de los resultados de la simulación podremos identificar y detectar errores antes de implementar el diseño en la FPGA. Por último, una vez que se hayan validado cada uno de los bloques funcionales se procederá a la integración en el sistema completo.

7.3. Funcionamiento de una UART

En esta práctica diseñaremos un transmisor/receptor serie asíncrono (*Universal Asynchronous Receiver-Transmitter*) que siga la norma RS-232. Este módulo nos permitirá comunicar nuestra placa con el puerto serie de la computadora. En la norma se definen dos tipos de terminales: DTE y DCE, donde DTE es el equipo terminal de datos (la computadora) y DCE es el de comunicación de datos, y que habitualmente es un MODEM y en nuestro caso es la placa de la FPGA.

Aunque en la norma se definen más señales, para establecer la comunicación sólo son imprescindibles tres señales: la línea de transmisión de datos (TxD), la de recepción (RxD), y la línea de masa (GND). Como puedes ver en la figura 7.1, la referencia se toma desde la computadora (el DTE). Por tanto, desde el punto de vista de la FPGA (DCE), la señal RxD es la que se transmite a la computadora, mientras que la señal TxD es la que se recibe de la computadora. Tendremos que tener cuidado porque esto nos puede confundir.

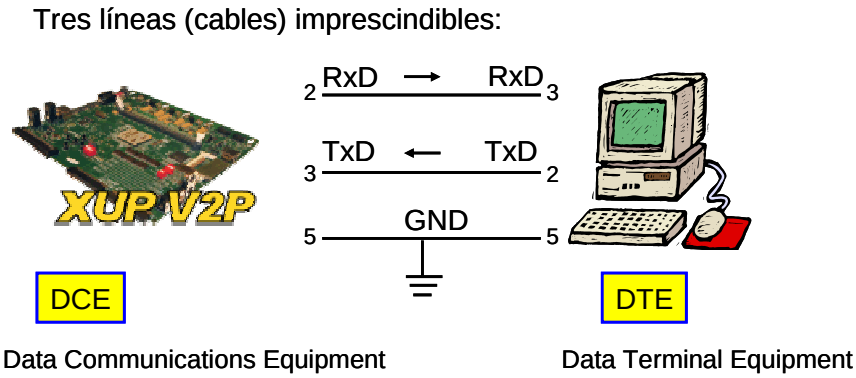


Figura 7.1:Esquema de la conexión RS-232

Para la conexión serie, las placas tienen un conector DB9 como el mostrado en la figura 7.2. El conector DB9 hembra está en el DCE y el macho en el DTE (computadora).

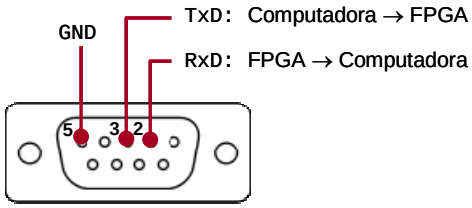


Figura 7.2: Conector DB9 hembra de la placa y los pines utilizados

Los pines de la FPGA que están conectados con el puerto RS-232 se muestran en la tabla 7.1. Aunque la placa XUPV2P tiene disponibles más pines que están incluidos en el protocolo, sin embargo sólo utilizaremos los dos imprescindibles[23] (mostrados en la tabla 7.1). Para evitar confusiones acerca de los nombres, los puertos del RS-232 los llamaremos como muestra la tabla 7.1, de modo que al incluir el prefijo FPGA indicamos que hacemos referencia a la FPGA. Por tanto, el puerto FPGA_TX representa el dato que envía la FPGA a la computadora, que en el protocolo RS-232 sería el RxD por estar referenciado al computador.

Puerto	Sentido	PIN	
		XUPV2P	Nexys2
FPGA_TX	FPGA → Computadora	AE7	P9
FPGA_RX	Computadora → FPGA	AJ8	U6

Tabla 7.1: Puertos del RS-232 que usaremos y los pines en las placas

Existen distintas velocidades de transmisión, que se definen en bits por segundo (bps) o baudios (921600, 460800, 230400, 115200, 57600, 38400, 19200, 9600, 4800, …). También se

[23] En la Nexys2 no hay más pines disponibles para la UART

puede variar el número de bits del dato que se envía, así como el envío de un bit de paridad y el número de bits de fin.

La línea serie permanece a nivel alto ('1') mientras no se envían datos. Cuando el transmisor va a empezar la transmisión, lo hace enviando un bit de inicio poniendo la línea a cero. Posteriormente se envían consecutivamente los bits del dato empezando por el menos significativo. Después del último bit de dato se envía el bit de paridad en caso de que se haya especificado. Por último, se cierra la trama con uno o dos bits de fin poniendo la línea a nivel alto. En la figura 7.3 se muestra el cronograma de un envío RS-232 con 8 datos (en este ejemplo: 11011101), un bit de paridad (en este caso par) y un bit de fin.

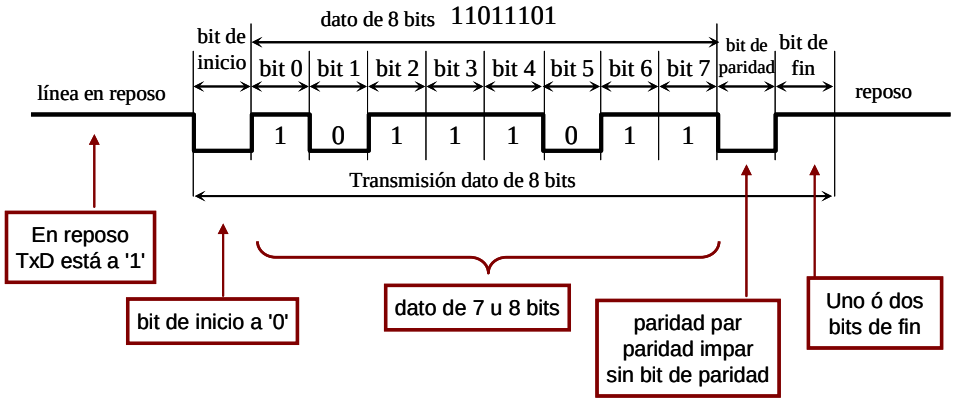


Figura 7.3: Trama de un envío en RS232 con 8 bits, bit de paridad y un bit de fin

La práctica consistirá en diseñar una UART compatible con RS-232 que envíe 8 bits de datos, que pueda recibir y transmitir simultáneamente (*full-duplex*), sin bit de paridad y con un bit de fin.

Para ir paso a paso, primero se realizará sólo el transmisor. Posteriormente se realizará el receptor, para luego unirlos en un único diseño.

7.4. Diseño del transmisor

Haremos un transmisor que tendrá los siguientes puertos (figura 7.4):

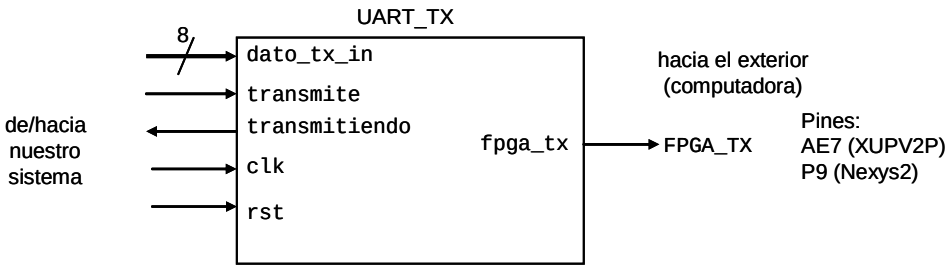


Figura 7.4: Entradas y salidas del transmisor

En la figura 7.4, los puertos de la izquierda son los que se relacionan con nuestro sistema (el que implementemos en la FPGA) y el puerto de la derecha (`fpga_tx`) envía el dato serie a la computadora. Antes de pasar a describir las señales, en la tabla 7.2 se incluyen unas constantes que se usarán en el diseño (recuerda el capítulo 6). Fíjate que las constantes comienzan por "c_" para que sea más fácil su identificación.

Constante	Tipo	Descripción
`c_on`	`std_logic`	Indica el nivel de la lógica. '1' para lógica positiva (Nexys2), y '0' para lógica negada (XUPV2P)
`c_freq_clk`	`natural`	Frecuencia del reloj. En nuestro caso: 10^8 para la XUPV2P y $5{\cdot}10^7$ para la Nexys2
`c_baud`	`natural`	Los baudios a los que va a transmitir la UART: 9600, 115200, ...

Tabla 7.2: Constantes de la UART para su configuración

Las especificaciones de los puertos son las siguientes:

Puerto	bits	I/O	Descripción
`rst`	1	I	Señal de reset asíncrono, su nivel dependerá de `c_on`
`clk`	1	I	Señal de reloj de la placa. La frecuencia del reloj se indica en la constante `c_freq_clk`.
`transmite`	1	I	Señal del sistema que ordena al módulo la transmisión del dato que se encuentra en `dato_tx_in`. La orden será de un único ciclo de reloj.
`dato_tx_in`	8	I	Dato que se quiere enviar, se proporciona cuando se activa `transmite` (cuando `transmite='1'`)
`transmitiendo`	1	O	Indica al sistema que el módulo está transmitiendo y por tanto no podrá atender a ninguna nueva orden de transmisión. Ignorará a `transmite` cuando `transmitiendo` esté a uno
`fpga_tx`	1	O	Trama que se envía en serie a la computadora, sigue el formato RS232

Tabla 7.3: Puertos del transmisor de la UART

Con estas especificaciones tenemos información suficiente para hacer el transmisor. Lo primero que tienes que hacer es crear un nuevo proyecto llamado `uart_tx`. Dentro del proyecto crea el paquete `UART_PKG` (recuerda la sección 6.3 para ver cómo se crean los paquetes) e incluye las constantes de la tabla 7.2. Es importante que en el paquete pongas comentarios sobre el significado y los valores de las constantes, y sobre todo, que comentes qué valores son los más habituales. En nuestro caso deberías de comentar qué valores tendría si se usase para la *XUPV2P* y para la *Nexys2*. Para mostrar un ejemplo de cómo se pueden poner estos comentarios, se ha añadido el código 7.1.

```
package UART_PKG is
   --------------------- declaracion de constantes ----------------------------------
   -- c_on: indica el tipo de logica de los pulsadores, interruptores y LEDS
   -- si es '1' indica que es logica directa -> PLACA NEXYS2
   -- si es '0' indica que es logica directa -> PLACA XUPV2P
   constant c_on        : std_logic := '0'; -- XUPV2P

   -- c_freq_clk: indica la frecuencia a la que funciona el reloj de la placa
   -- para la Nexys2 el reloj va a  50MHz -> 5*10**7;
   -- para la XUPV2P el reloj va a 100MHz -> 10**8;
   constant c_freq_clk  : natural   := 10**8; -- XUPV2P

   -- c_baud: indica los baudios a los que transmite la UART, valores
   -- tipicos son 9600, 19200, 57600, 115200
   -- Este valor depende de la conexion establecida con la computadora
   constant c_baud      : natural   := 115200;

end UART_PKG;
```

Código 7.1: Paquete con la declaración de constantes

Ahora crea la entidad `UART_TX`. En la entidad incluye los puertos de la tabla 7.3 y no te olvides de incluir la referencia al paquete en la cabecera de la UART[24].

El siguiente paso es diseñar el transistor de la UART por dentro, es decir, tenemos que crear la arquitectura. Con lo que ya sabemos, te recomendamos que pienses cómo lo harías antes de pasar a ver la solución propuesta. Existen muchas alternativas de diseño para una misma funcionalidad y la que se propone aquí no es la única ni tampoco tiene que ser la mejor.

Según las especificaciones del funcionamiento del protocolo RS-232 podemos pensar que necesitaremos:

- Un divisor de frecuencia para generar una señal periódica con la frecuencia indicada en los baudios.
- Un registro que guarde el dato que vamos a enviar, y que lo vaya desplazando según el bit que estemos enviando
- Un selector (multiplexor) que envíe el bit de inicio, el bit de fin, los bits de datos o que mantenga en reposo la línea.
- Un bloque de control que indique al resto de bloques en que estado estamos, es decir, qué es lo que toca enviar: bit de inicio, de fin, bits de datos o reposo.

Así que en una primera versión hemos identificado estos cuatro bloques que podrían organizarse como se muestra en la figura 7.5. Normalmente, estas primeras versiones las tendremos que modificar conforme vamos realizando el circuito. Aún así, cuanto más lo pensemos al principio, menos modificaciones tengamos que hacer, y posiblemente perdamos menos tiempo.

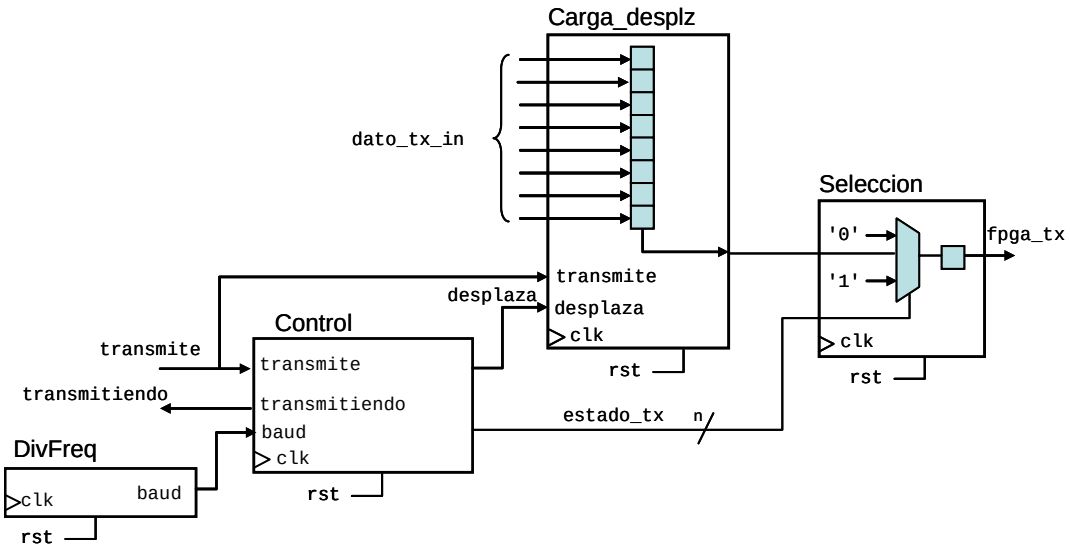


*Figura 7.5: Diagrama de bloques **preliminar** del transmisor de la UART*

A continuación explicaremos cada uno de los bloques.

7.4.1. Divisor de frecuencia

Este bloque se corresponde con el bloque `DivFreq` de la figura 7.5. Este bloque generará la señal que indica cuándo ha pasado el intervalo de tiempo correspondiente a cada bit de la

[24] Recuerda el código 6.10: `library WORK; use WORK.UART_PKG.ALL;`

trama de envío (figura 7.3). Esta señal tendrá una frecuencia determinada por la constante `c_baud` y que se corresponde con los baudios a los que se comunicará la UART. En caso de que te hayas olvidado de los contadores y divisores de frecuencia, te recomendamos que repases el capítulo 6 de las prácticas de ED2 [17mach].

Este diseño lo queremos hacer configurable, de modo que si en un determinado momento queremos cambiar los baudios de la UART o si usamos una placa con distinta frecuencia de reloj, no tengamos que rehacer el circuito, sino que nos baste con cambiar el valor de las constantes (tabla 7.2 y código 7.1).

Realizar un diseño configurable es más complicado al principio y hace que se tarde más en diseñar. Sin embargo, a la larga hace ahorrar mucho tiempo porque no hace falta rediseñar. Además el diseño configurable suele ser más seguro, ya que si tenemos que rediseñar un circuito podemos cometer nuevos errores.

En nuestro caso, el diseño configurable también nos puede hacer ahorrar tiempo de simulación, ya que para simular se pueden poner frecuencias mayores de envío de la UART, y así no tener que esperar los largos tiempos de simulación.

Para facilitar la comprensión y el diseño, antes de pensar en el diseño configurable, veamos cómo haríamos el diseño con **valores concretos**.

Por ejemplo, a partir del reloj de la placa XUPV2P de 100 MHz (`clk`) supongamos que queremos transmitir con frecuencia de 9600 Hz (`baud`). Por tanto, la señal `c_baud` tendrá un periodo de 104,167 μs, y la crearemos de modo que esté un único ciclo de reloj a uno y el resto del tiempo a cero. La figura 7.6 representa los cronogramas de las señales de entrada (`clk`) y salida (`baud`).

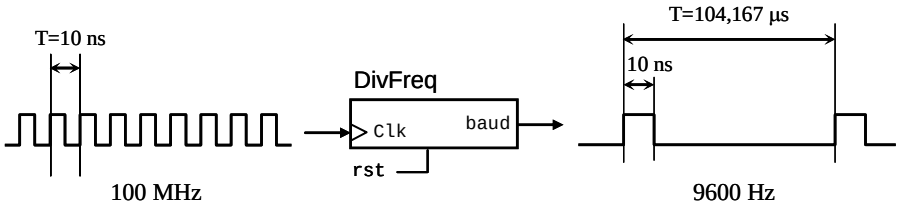


Figura 7.6: Divisor de frecuencia de 100 MHz a 9600 Hz

Para diseñar el divisor de frecuencia se divide la frecuencia de entrada entre la frecuencia de salida (100MHz/9,6kHz = 10416,67→10417) y el número resultante nos dará la cuenta necesaria para obtener la frecuencia de salida. Haciéndolo de manera inversa, 10417 cuentas de 10 ns (`clk`) nos da un periodo de 104,17μs, que es una buena aproximación de la frecuencia que queremos. Esto es, queremos 9600 Hz y obtenemos 9599,69 Hz.

Para la creación de este bloque habría que tener en cuenta lo explicado referente a los rangos numéricos del apartado 4.2, y en vez de crear un contador con rango de 0 a 10416 que no tiene ninguna correspondencia física, lo haríamos con el rango que delimite el ancho del bus. Como $\log_2(10416) = 13{,}35$; necesitaríamos 14 bits para el contador.

Aunque en este ejemplo no es tan importante contar un ciclo más o menos, ya que 10 ns es despreciable frente a 104,167 μs, en general hay que ser cuidadosos para hacer un contador que cuente el número de ciclos que queremos. Conviene dibujar el cronograma del circuito (figura 7.7) y posteriormente diseñarlo.

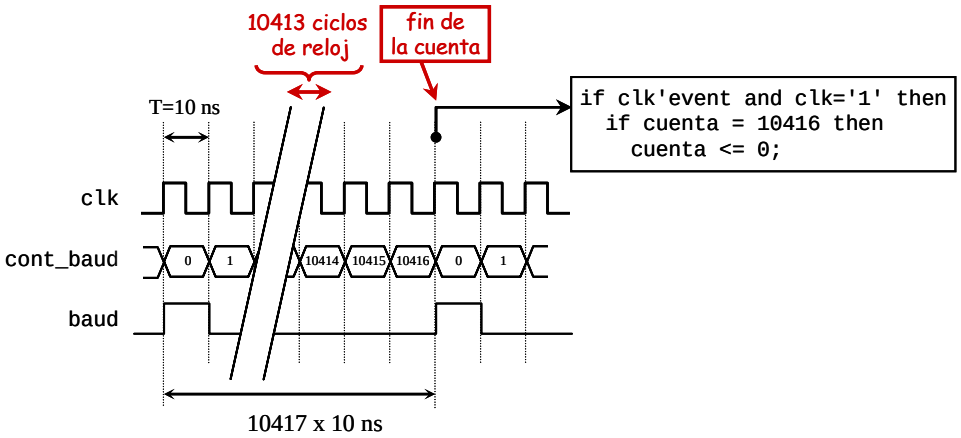


Figura 7.7: Cronograma del contador del divisor de frecuencia en el final de la cuenta

A partir del cronograma podemos ver que una vez que `cont_baud` valga 10416 es cuando debemos inicializar el contador. Como las transiciones ocurren durante el flanco activo de reloj (`clk'event and clk='1'`) podemos confundirnos con los valores que se evalúan para cada señal en el evento de reloj, ¿qué valor hemos de poner en la comparación, el anterior o el posterior a la transición? En nuestro caso, ¿comparamos con 10416 o con 10417 para iniciar el contador?

La figura 7.8 muestra el cronograma ampliado. Podemos apreciar que es a partir del flanco del reloj cuando las señales registradas (`cont_baud` y `baud`) cambian de valor, por tanto, se comparan los valores que las señales tienen justo antes del flanco de reloj. Por tanto, en nuestro caso, comparamos con 1416.

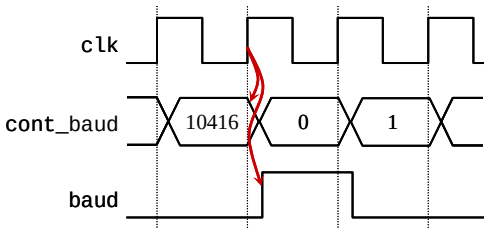


Figura 7.8: Cronograma en la transición a cero

Ahora que ya sabemos cómo hacer el diseño con valores concretos, veamos cómo se haría **configurable** con las constantes.

En el ejemplo anterior, para calcular el fin de cuenta dividimos la frecuencia de reloj de la placa entre la velocidad de transmisión (100MHz/9,6kHz = 10416,67→10417). Por lo tanto, en vez de hacer que el fin de cuenta sea hasta 10417, usaremos una constante (`c_fin_conta_baud`) que tenga el valor resultante de la división entre la constante que indica la frecuencia y la constante que indica la velocidad de transmisión. Esta constante se muestra en el código 7.2 y la deberemos incluir en el paquete. El operador "`/`" realiza la división entera. Observa que se le resta uno porque la cuenta empieza desde cero.

```
constant c_fin_cont_baud  : natural := c_freq_clk/c_baud - 1;
```

Código 7.2: Constante para el cálculo del fin de cuenta de divisor de frecuencia de la UART

Con esto, en la arquitectura del transmisor podríamos declarar la señal de la cuenta con el rango dependiente de la constante `c_fin_cont_baud`.

```
signal   cont_baud     : natural range 0 to c_fin_cont_baud;
```

Código 7.3: Rango de la señal cuenta basado en la constante de fin de cuenta

Sin embargo, con el código 7.3 cometemos el mismo fallo que mostramos en el apartado 4.1 (recuerda los códigos 4.1 y 4.3), en donde el rango no es potencia de dos. Esto lo podríamos solucionar asegurándonos que la señal nunca sobrepase el fin de cuenta,

aunque una alternativa más segura es usar una función que calcule el rango. Esto se explicará en el subapartado siguiente.

7.4.1.1. Uso de funciones

Queremos calcular cuántos bits necesitamos para la señal `cont_baud`, para ello podemos crear una función que nos calcule el logaritmo en base dos del valor del fin de cuenta (`c_fin_cont_baud`). En VHDL una función puede tener varios argumentos de entrada y devuelve una salida.

En VHDL se pueden declarar funciones en la parte declarativa de la arquitectura (antes del `begin`), sin embargo nosotros la pondremos en el paquete que hemos creado. De esta manera la podremos usar en el resto de módulos del diseño.

Por tanto, se ha de incluir la declaración de la función (código 7.4) en el paquete que hemos creado (código 7.1). Es decir, copia el código 7.4 en el paquete `UART_PKG`. Observa los comentarios de la función, es importante que incluyas comentarios para saber cómo se usa la función sin tener que mirarte el código para descubrir su funcionamiento.

```
-------------------------- funcion: log2i -------------------------------------
-- Descripcion: funcion que calcula el logaritmo en base 2 de un numero entero
--              positivo. No calcula decimales, devuelve el entero menor o igual
--              al resultado - por eso la i (de integer) del nombre log2i.
--              P. ej: log2i(7) = 2, log2i(8) = 3.
-- Entradas:
--   * valor: numero entero positivo del que queremos calcular el logaritmo en
-- Salida:
--   * devuelve el logaritmo truncado al mayor entero menor o igual que el resultado
function log2i (valor : positive) return natural;
```

Código 7.4: Declaración de una función para el cálculo del logaritmo

Cuando una función se declara en un paquete, el cuerpo de la función (*function body*) se tiene que incluir en el cuerpo del paquete (*package body*). El cuerpo del paquete lo podemos crear a continuación del paquete (*package*).

Para calcular el logaritmo, la función `log2i` tendrá que dividir por dos de manera sucesiva hasta que el resultado sea cero (realizando la división entera). El número de divisiones que hayamos realizado nos indicará el número de bits necesarios.

En el código 7.5 se ha incluido el cuerpo del paquete `UART_PKG` y dentro se ha incluido el cuerpo de la función `log2i`. Observa que se han incluido comentarios que explican cómo realiza las operaciones internamente. Quizá no sea necesario tanto detalle, pero es importante para facilitar la labor en el futuro si tuvieses que hacer correcciones o modificaciones. Copia este código e inclúyelo después del paquete.

```
Package body UART_PKG is
 -------------------- Funcion log2i -------------------------------------
 -- Ejemplos de funcionamiento (valor = 6, 7 y 8)
 --  * valor = 6            |  * valor = 7          |  * valor = 8
 --      tmp = 6/2 = 3      |     tmp = 7/2 = 3     |     tmp = 8/2 = 4
 --      log2 = 0           |     log2 = 0          |     log2 = 0
 --    - loop 0: tmp 3 > 0  |   - loop 0: tmp 3>0   |   - loop 0: tmp 4>0
 --      tmp = 3/2 = 1      |     tmp = 3/2 = 1     |     tmp = 4/2 = 2
 --      log2 = 1           |     log2 = 1          |     log2 = 1
 --    - loop 1: tmp 1 > 0  |   - loop 1: tmp 1 > 0 |   - loop 1: tmp 2 > 0
 --      tmp = 1/2 = 0      |     tmp = 1/2 = 0     |     tmp = 2/2 = 1
 --      log2 = 2           |     log2 = 2          |     log2 = 2
 --    - end loop: tmp = 0  |   - end loop: tmp = 0 |   - loop 2: tmp 1 > 0
 --  * return log2 = 2      | * return log2 = 2     |     temp = 1/2 = 0
 --                                                 |     log2 = 3
 --                                                 |   - end loop: tmp = 0
 --                                                 | * return log2 = 3
  function log2i (valor : positive) return natural is
    variable tmp, log2: natural;
  begin
    tmp := valor / 2;  -- division entera, redondea al entero inmediatamente menor o =
    log2 := 0;
    while (tmp /= 0) loop
      tmp := tmp/2;
      log2 := log2 + 1;
    end loop;
    return log2;
  end function log2i;
end UART_PKG;
```

Código 7.5: Cuerpo del paquete `UART_PKG` y cuerpo de la función `log2i`

Observa que dentro de la función se han declarado variables, que son distintas que las señales. A diferencia de las variables, las señales no se pueden declarar dentro de una función. La asignación de las variables se realiza con el operador ":=". Las variables toman su valor con la asignación y no como ocurre con las señales en los procesos que toman el valor al salir de proceso.

Fíjate que la función devuelve un número menor al número de bits necesario, por ejemplo, para representar un 6 necesitamos 3 bits y no 2 (que será lo que devuelva `log2i`). Si queremos representar un entero, tendremos que sumar uno al resultado, sin embargo, para los tipos `unsigned` nos viene bien así, porque el rango incluye el cero. Para facilitar la su comprensión, en la figura 7.9 se detalla este concepto.

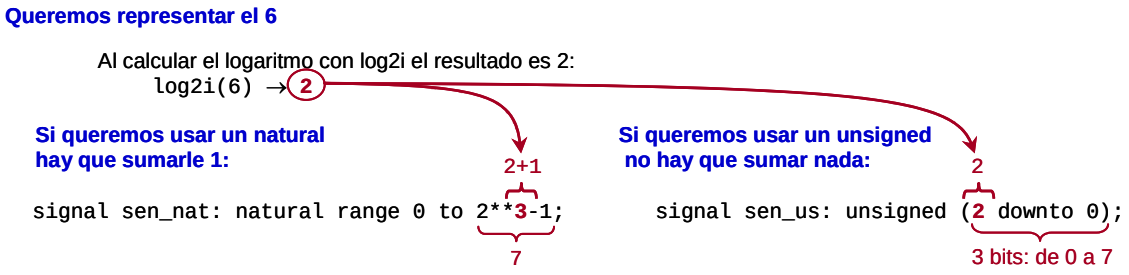


Figura 7.9: Uso del valor devuelto por la función `log2i` para la obtención del rango de un natural y un unsigned

En el código 7.6 se muestran un ejemplo de cómo se declararían dos señales (`natural` y `unsigned`) a partir del resultado devuelto por la función `log2i`.

```
 constant log2_6  : natural := log2i(6);       -- para representar 6, log2i devuelve 2
 signal sen_nat: natural range 0 to (2**(log2_6+1))-1; -- si es entero hay que sumar 1
 signal sen_us : unsigned (log2_6 downto 0);           -- si es unsigned dejamos igual
```

Código 7.6: Ejemplo de la utilización del resultado de la función `log2i` para los rangos de las señales

Así pues, declararemos una constante que represente el número de bits necesarios para el contador (en realidad el número de bits menos 1). Esta constante la pondremos en el paquete. Tiene que estar después de la declaración de la función (código 7.4) ya que utiliza dicha función[25] (lo mejor es que la declaración de la función la pongas al principio del paquete). Esta declaración está en el código 7.7.

```
-- el numero de bits (nb) menos 1 necesarios para representar c_fin_cont_baud
constant c_nb_cont_baud : natural := log2i(c_fin_cont_baud);
```

Código 7.7: Constante que representa el número de bits menos uno del contador del divisor de frecuencia

Ahora ya podemos declarar la señal del contador en la arquitectura. El código 7.8 muestra la declaración de la señal para el caso de tipo `unsigned`.

```
signal cont_baud : unsigned (c_nb_cont_baud downto 0);  -- unsigned: dejamos igual
```

Código 7.8: Declaración de la seña , para el caso de tipo `unsigned`, que cuenta con el rango determinado por la constante calculada

Si quisiésemos declararla como natural, podríamos usar la declaración del código 7.9. Se ha incluido como ejemplo, ya que en este diseño diseñaremos el contador con tipo `unsigned`. Fíjate que en el código 7.9 se ha sumado 1 a la constante `c_nb_cont_baud`, y luego se le resta 1 al resultado de la exponenciación, no es que se le estemos restando 1 otra vez al exponente. Esto es por la precedencia de los operadores, la exponenciación tiene mayor precedencia que la resta.

```
signal cont_baud : natural range 0 to 2**(c_nb_cont_baud+1)-1;  -- entero: sumar 1
```

Código 7.9: Declaración de la señal, para el caso de tipo natural, con el rango determinado por la constante calculada

Así que con la señal declarada, ya podemos realizar el divisor de frecuencia. Intenta hacerlo tú mismo sin mirar el código 7.10, de todos modos, como se han introducido muchas constantes y conceptos nuevos, hemos incluido una posible solución para que la compares con tu divisor de frecuencia. Es importante que antes de mirar la solución diseñes la tuya, pues conforme avancemos en el libro, se va a reducir el código de ejemplo.

```
P_DivFreq: Process(rst,clk)
begin
  if rst = c_on then
    cont_baud <= (others => '0');  -- todo a cero
    baud <= '0';
  elsif clk'event and clk='1' then
    baud <= '0';  -- por defecto, siempre a cero
    if cont_baud = c_fin_cont_baud then
      baud <= '1';  -- avisamos de fin de cuenta
      cont_baud <= (others => '0');
    else
      cont_baud <= cont_baud + 1;
    end if;
  end if;
end process;
```

Código 7.10: Proceso del divisor de frecuencia del transmisor de la UART

7.4.1.2. Operaciones con constantes y con señales en VHDL

En el capítulo 8 veremos que muchas de las operaciones que se consideran simples, como la división, no tienen una implementación directa en hardware. Por ejemplo, para realizar

[25] **Importante**: En algunos simuladores (por ejemplo, la versión 6.0 de Modelsim), no se puede usar una función antes de tener declarado su cuerpo. Si tienes este problema consulta el apartado 7.4.1.3

una división tenemos que diseñar un módulo VHDL que describa el hardware que la implemente, y ya veremos que no hay única manera de implementar un divisor.

A raíz de esto nos podríamos preguntar si lo que hemos hecho en el apartado anterior es correcto, ya que hemos calculado el logaritmo de un número mediante divisiones sucesivas. Por tanto podríamos preguntarnos ¿podemos realizar estas divisiones en VHDL tan inocentemente? la respuesta es *según con qué operandos*.

Al sintetizar en VHDL se pueden realizar cálculos complejos (divisiones, exponenciaciones,...) en el caso de que esas operaciones se realicen durante el proceso de síntesis (antes de bajarlo a la FPGA). Esto implica que **los operandos tienen que ser constantes**. Es decir, podemos realizar el logaritmo de 234, pero no podemos implementar el logaritmo de la señal `X` (en realidad se podría, pero tendríamos que describir en VHDL el módulo sintetizable que lo implementase).

Así que la función `log2i` la podemos utilizar para calcular el logaritmo de constantes, pero no la podremos utilizar para implementarla en el circuito y calcular el logaritmo de señales, pues van a cambiar.

Durante la síntesis, la herramienta ISE es la que calcula los valores resultantes de aplicar la función `log2i` a las constantes. Es decir, no es el circuito VHDL el que implementa la función y calcula el resultado, sino el sintetizador. Si fuesen señales (que cambian de valor) no se podría calcular el valor durante la síntesis, y por tanto habría que implementar en hardware el operador, y por lo tanto no se podría usar la función.

7.4.1.3. Uso de funciones dentro del mismo paquete

En algunos simuladores (por ejemplo, la versión 6.0 de Modelsim), no se puede usar una función antes de tener declarado su cuerpo. Esto hace que no podamos usar una función para constantes declaradas en el mismo paquete. Este caso se muestra en el código 7.11.

```
package UART_PKG is
 function log2i (valor : positive) return natural;
 constant c_nb_cont_baud : natural := log2i(c_fin_cont_baud);
end UART_PKG;

Package body UART_PKG is
  function log2i (valor : positive) return natural is
    variable tmp, log2: natural;
  begin
    tmp := valor / 2;  -- division entera, redondea al entero menor
    log2 := 0;
    while (tmp /= 0) loop
      tmp := tmp/2;
      log2 := log2 + 1;
    end loop;
    return log2;
  end function log2i;
end UART_PKG;
```

Código 7.11: El uso de funciones dentro del mismo paquete da problemas en algunas herramientas

Esto no es un problema para el ISE, pero si queremos usar la versión 6.0 del Modelsim, para solucionar esto tenemos que usar dos paquetes, uno de los paquetes tendrá todas las declaraciones de funciones (`UART_PKG_FUN`) y en ese fichero incluiremos el cuerpo del paquete con todos los cuerpos de las funciones.

En el segundo paquete incluiremos todas las declaraciones de constantes (`UART_PKG`). El paquete que tiene las declaraciones de constantes tendrá que referenciar el paquete que de

las funciones[26], mientras que el resto de unidades harán referencia al paquete de las constantes, a no ser que necesiten usar alguna de las funciones y entonces también tendrán que hacer referencia a los dos paquetes.

Existe una alternativa para el Modelsim, que es usar declaraciones diferidas de constantes (*deferred constant*). Esto consiste en declarar la constante en el paquete sin darle valor y volver a declarar la constante dándole su valor en el cuerpo del paquete, después del cuerpo de la función. En el código 7.12 se muestra cómo solucionar el código 7.11.

```
package UART_PKG is
 function log2i (valor : positive) return natural;

  -- declaracion diferida de constante (deferred constant), sin valor
 constant c_nb_cont_baud : natural;
end UART_PKG;

Package body UART_PKG is
  function log2i (valor : positive) return natural is
    variable tmp, log2: natural;
  begin
    tmp := valor / 2;  -- division entera, redondea al entero menor
    log2 := 0;
    while (tmp /= 0) loop
      tmp := tmp/2;
      log2 := log2 + 1;
    end loop;
    return log2;
  end function log2i;

  -- se le da el valor a la constante en el cuerpo del paquete
  constant c_nb_cont_baud : natural := log2i(c_fin_cont_baud);

end UART_PKG;
```

Código 7.12: Uso de constantes diferidas para evitar problemas por usar funciones dentro del mismo paquete (código 7.11). No vale para el ISE

Por desgracia, este método no lo acepta el ISE. Con esto puedes ver que muchas veces el diseñador está limitado a las herramientas que utilice. En el caso que acabamos de ver, lo que acepta una, no lo acepta la otra y viceversa.

*7.4.1.4. Más funciones **

Este apartado te lo puedes saltar si estás cansado de funciones, constantes y demás. Es simplemente un comentario para los más atentos.

Quizás te has fijado que en el código 7.2, el resultado de realizar la división entera de `c_freq_clk/c_baud - 1` da 10415 y nosotros habíamos calculado que la constante debería ser 10416 por el redondeo, ya que la división da 10416,67 y habíamos redondeado a 10417 (y luego le habíamos restado uno porque se cuenta desde cero).

Como el operador "`/`" realiza la división entera truncando al entero inferior inmediato, nosotros podríamos crear una función entera con redondeo al entero más cercano. Esta función la podemos incluir en el paquete y calcular el fin de cuenta (cambiando el código 7.2) con esta función en vez de con el operador "`/`". Aunque incluimos la función en el código 7.13, intenta hacerla por ti mismo antes de mirar el código.

[26] Recuerda el código 6.10: `library WORK; use WORK.UART_PKG_FUN.ALL;`

```
function div_redondea (dividendo, divisor: natural) return integer is
    variable division : integer;
    variable resto    : integer;
begin
  division := dividendo/divisor;
  resto    := dividendo rem divisor; -- rem: calcula el resto de la division entera
  if (resto > (divisor/2)) then
    division := division + 1;
  end if;
  return (division);
end;
```

Código 7.13: Función para el redondeo

7.4.2. Bloque de control

El bloque de control es el encargado dirigir al resto de bloques. Para realizar el control podríamos pensar en cuatro estados: inicial (reposo), envío del bit de inicio, envío de los 8 bits de datos y el envío del bit de fin (en caso de que hubiese bit de paridad habría que incluir un estado más). La figura 7.10 muestra una versión preliminar del diagrama de estados.

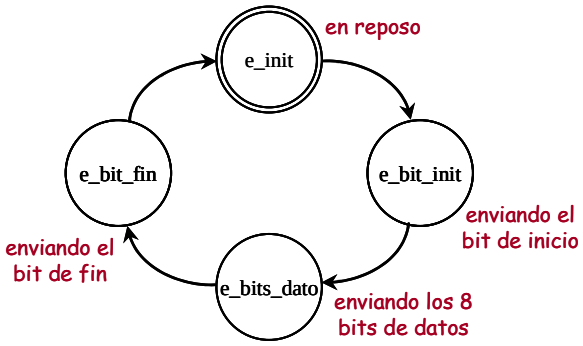


Figura 7.10: Diagrama de estados preliminar del transmisor de la UART

Hay otras opciones alternativas a la figura 7.10, por ejemplo incluir un estado para cada bit. Como ya sabes, una misma funcionalidad se puede implementar de muchas maneras. Vamos a examinar qué harían cada uno de estos estados:

- Estado inicial (**`e_init`**): en este estado inicial el sistema está en reposo esperando la orden de transmitir. Por tanto, la línea de transmisión estará a uno (`fpga_tx='1'`), ya que es el valor inactivo. Cuando la señal `transmite` valga '`1`' será el indicativo de que se quiere transmitir un dato, y por tanto se pasará a enviar el bit de inicio: `e_bit_init`. En este momento se dará la orden de cargar el dato (señal `cargadato`) que se quiere enviar (`dato_tx_in`) en el registro de desplazamiento. Simultáneamente debemos sincronizar el contador del divisor de frecuencia, para ello podríamos deshabilitar el contador durante el estado inicial para que no cuente. Esta señal de habilitación (`en_divfreq`) no la habíamos contemplado en la versión preliminar del diagrama de bloques (figura 7.5).
- Envío de bit de inicio (**`e_bit_init`**): en este estado se está enviando el bit de inicio, poniendo la línea de transmisión a cero (`fpga_tx='0'`). Se saldrá de este estado cuando haya pasado el periodo de tiempo correspondiente a un bit. Este tiempo viene determinado por la señal `baud`, generada por el divisor de frecuencia (apartado 7.4.1). Después de este tiempo se pasará a enviar los bits de dato.
- Envío de los bits del dato (**`e_bits_dato`**): en este estado se envían los 8 bits del dato. Una alternativa a esta opción sería poner un estado para cada bit, sin embargo, usaremos un estado para los 8 bits y, mediante un contador, llevaremos la cuenta del número de bits que se han enviado. Para enviar cada bit, el control generará la señal `desplaza` que le indica al registro de desplazamiento que tiene desplazar sus bits. Cuando se hayan enviado/contado los 8 bits el contador generará una señal (`fin_cont8bits`) que indicará que hay que pasar a enviar el bit de fin y por lo tanto cambiaremos de estado (si quisiésemos implementar el bit

de paridad, iría ahora). Por lo dicho anteriormente, en el diagrama preliminar de la figura 7.5 habrá que añadir un bloque que cuente los bits.

- Envío del bit de fin (**`e_bit_fin`**): En este estado se envía el bit de fin, poniendo la línea de transmisión a uno (`fpga_tx='1'`). Se saldrá de este estado cuando haya pasado el periodo correspondiente a un bit (ponemos un único bit de fin), este periodo de tiempo lo indica la señal baud.

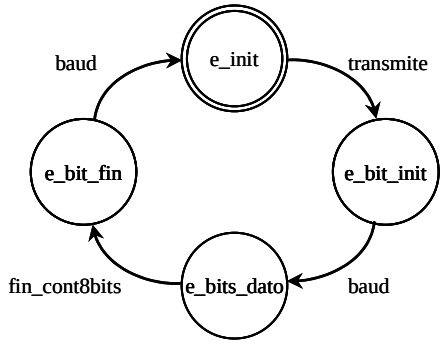


Figura 7.11: Diagrama de estados con la indicación de las señales que hacen cambiar de estado

En la figura 7.11 se muestra el diagrama de estados con la indicación de las señales que hacen cambiar de estado.

Como hemos visto en el análisis de cada uno de los estados, nos hacen falta más señales e incluso un bloque nuevo respecto a lo que inicialmente vimos en la figura 7.5. Así que el nuevo diagrama de bloques podría quedar como muestra la figura 7.12, donde se incluye el bloque contador de los bits que se van enviando, y las señales `en_divfreq`, `cargadato` y `fin_cont8bits`. Como siempre, existen multitud de propuestas que también serían válidas.

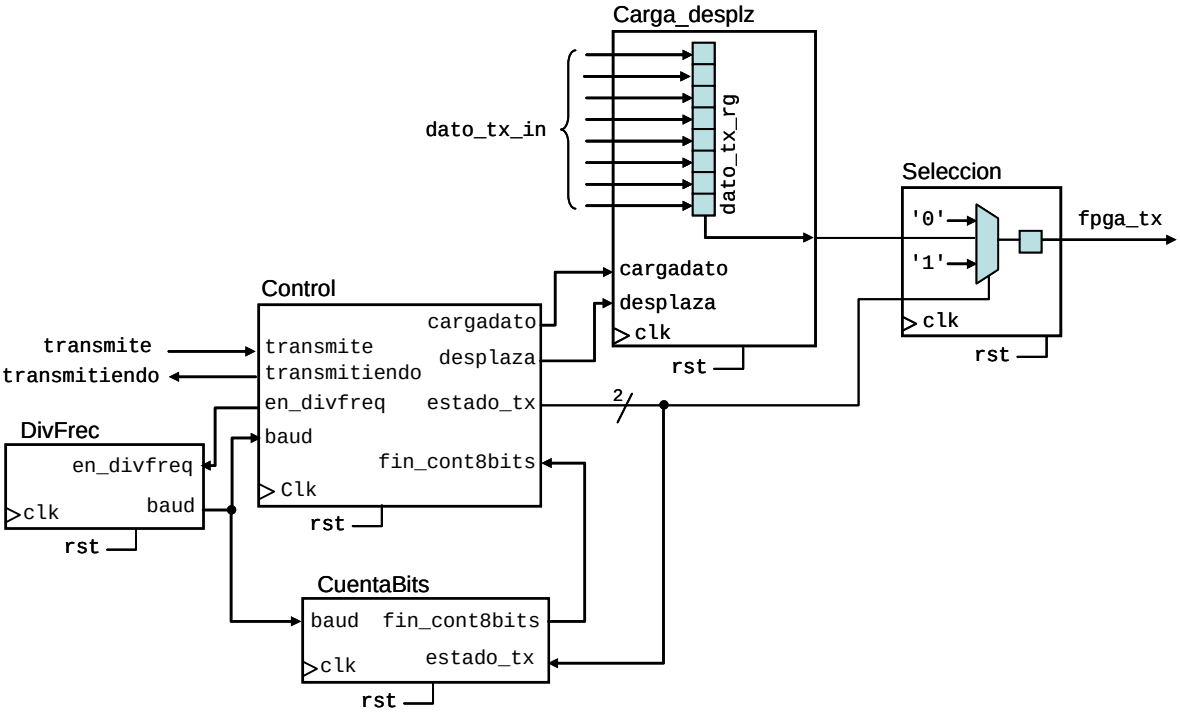


Figura 7.12: Diagrama de bloques definitivo del transmisor

Observa que ahora el divisor de frecuencia tiene una señal de habilitación (`en_divfreq`), así que tienes que modificar el código 7.10 de modo que incluya la habilitación del contador. Si no te acuerdas cómo se hace, un contador con habilitación lo puedes ver en el capítulo 16 del manual de ED2 (código 16.2).

En la tabla 7.4 se muestra la tabla de estados, entradas y salidas. Antes de mirarla intenta realizarla por ti mismo. Puede ser que algunas salidas no te salgan igual, especialmente

las de las transiciones, en algunas no tiene que ser un error que no estén iguales que la tabla 7.4.

Estado Actual	Entradas			Estado siguiente	Salidas			
	transmite	baud	fin_cont8bits		cargadato	desplaza	en_divfreq	transmitiendo
e_init	**0**	X	X	e_init	0	0	0	0
e_init	**1**	X	X	e_bit_init	1	0	0	0
e_bit_init	X	**0**	X	e_bit_init	0	0	1	1
e_bit_init	X	**1**	X	e_bits_dato	0	0	1	1
e_bits_dato	X	X	**0**	e_bits_dato	0	baud	1	1
e_bits_dato	X	X	**1**	e_bit_fin	0	0	1	1
e_bit_fin	X	**0**	X	e_bit_fin	0	0	1	1
e_bit_fin	X	**1**	X	e_init	0	0	1	1

Tabla 7.4: Tabla de estados, entradas y salidas del transmisor de la UART

Fíjate que la salida `desplaza`, se pone a uno cuando estando en el estado `e_bits_dato` la señal `baud` se pone a uno. Esto es porque cuando se están enviando los bits de dato hay que desplazar el registro de desplazamiento para cambiar el bit que se envía, esto se debe hacer cuando transcurre el periodo de tiempo marcado por la señal `baud`.

A partir de la tabla 7.4, el diagrama de la figura 7.11 y las explicaciones anteriores, quizá puedas realizar el módulo de control. Si no, repasa los capítulos 9 y posteriores del manual de ED2 [17mach].

También debes hacer el contador `CuentaBits` (figura 7.12). Fíjate que este contador debe funcionar solamente estando en el estado `e_bits_dato` y que la señal `fin_cuenta8bits` se debe de activar cuando se haya terminado de transmitir el octavo bit. Aquí debes de tener en cuenta la sincronización y evitar contar de más o de menos. Es muy probable que necesites simular para corregir algún error.

7.4.3. Registro de desplazamiento y multiplexor

Estos bloques se han visto en el manual de ED2 [17mach]. El registro de desplazamiento se explica en el apartado 7.1 y el multiplexor en el apartado 3.1 de dicho manual, además tienes ejemplos de multiplexores en el apartado 2.3 de este manual.

La salida del multiplexor la puedes pasar por un biestable y que sea la salida del biestable la que corresponda con la señal `fpga_tx`, ya que es una buena práctica de diseño registrar las salidas del diseño.

Importante: si registras la salida del multiplexor, como se recomienda en el párrafo anterior, debes tener en cuenta que en reposo, la línea de transmisión del puerto serie debe estar a '`1`', por lo tanto, al resetear, la señal `fpga_tx` la debes poner a '`1`' y no a '`0`' como suele ser más habitual. En otro caso, al resetear parecería que nuestro transmisor fuese a iniciar una transmisión.

Una vez que hayas realizado el diseño, comprueba que la sintaxis esté bien (*Check Syntax*), incluso prueba a sintetizarlo (*Synthesize - XST*), pero antes de implementarlo en la FPGA convendrá simularlo.

7.5. Simulación del transmisor

El transmisor que hemos realizado tiene cierta complejidad, y a diferencia de los circuitos que hemos realizado hasta ahora, no es fácil de comprobar si lo hemos diseñado bien. En los anteriores circuitos disponíamos de los leds para ver el comportamiento del circuito, ahora no tenemos los leds, y no es fácil comprobar si la transmisión se hace correctamente y cumple los especificados del protocolo RS-232. Así que es prácticamente imprescindible simular el circuito antes de implementarlo en la FPGA. En el capítulo 5 vimos los pasos para simular un circuito, también se explica en el capítulo 8 del manual de prácticas de ED2 [17mach]. Ahora vamos a seguir los mismos pasos, pero por las características de nuestro circuito tendremos que realizar un banco de pruebas de mayor complejidad.

La realización de un buen banco de prueba es una tarea muy importante en el diseño, incluso en diseños complejos puede llevar más tiempo la depuración que el propio diseño. Muchas veces el banco de pruebas y el diseño lo realizan personas diferentes para evitar que el diseñador realice el banco de pruebas en base a lo que ha diseñado o según su interpretación de las especificaciones del circuito. Un buen banco de pruebas debe comprobar que el diseño hace lo que debe hacer y no hace lo que no debe hacer.

A veces la complejidad del banco de pruebas es muy grande y se crean modelos de componentes para la simulación. La descripción de estos modelos puede ser muy diferente a los modelos que se sintetizan, ya que para simulación se acepta todo el conjunto del VHDL, mientras que para síntesis sólo se acepta un conjunto restringido. Para el banco de pruebas del transmisor recurriremos a este tipo de modelos.

Para empezar a realizar el banco de pruebas crearemos una nueva fuente de tipo *VHDL TestBench* y la llamamos `tb_uart_tx.vhd`. Este banco de pruebas estará asociado a la entidad `uart_tx` que ya hemos creado.

Lo primero que debes hacer es poner los mismos paquetes que pusiste en el transmisor de la UART, incluyendo la referencia al paquete que creamos en el código 7.1.

Ya que cada placa tiene un reloj con distinta frecuencia, en vez de crear un banco de pruebas distinto para cada placa, como hicimos en los códigos 5.1 y 5.2, crearemos un proceso de reloj dependiente de la constante de la frecuencia de reloj `c_freq_clk`.

Así que vamos a crear una constante con el periodo del reloj, para que pueda ser un número entero lo pondremos en nanosegundos, multiplicándolo por 10^9. La declaración de la constante se ha puesto en el código 7.14 y tendrás que incluirla en el paquete del código 7.1.

```
constant c_period_ns_clk : natural := 10**9/c_freq_clk;
```

Código 7.14: Constante con el periodo del reloj en nanosegundos

Con esta constante podemos crear el mismo proceso de reloj para las dos placas, ya que está hecho en función de la constante del periodo `cs_period_ns_clk`. Fíjate en el código 7.15 que como en la sentencia "`wait for`" hay que indicar un tiempo, hemos tenido que multiplicarla por un nanosegundo.

```
P_Clk: Process
begin
  clk <= '0';
  wait for (c_period_ns_clk / 2) * 1 ns;
  clk <= '1';
  wait for (c_period_ns_clk / 2) * 1 ns;
end process;
```

Código 7.15: Modificación del proceso del reloj para que su frecuencia dependa de constantes

Ahora crea el proceso de reset. Créalo de manera similar a los códigos 5.3 y 5.4 pero hazlo usando las constantes `c_on` y `c_off` (código 6.9) de manera que no tengas que usar un proceso distinto dependiendo de la placa que estés usando.

Fíjate en la declaración de señales del banco de pruebas. Observa que las señales tienen asignado un valor inicial. Quizá recuerdes que en síntesis no tiene sentido dar un valor inicial a las señales, pues sólo se puede dar valor inicial a las señales que forman biestable y éstos se inicializan con el reset. El resto de señales que no forman biestable reciben el valor de otras señales, y su valor inicial será dependiente de esas otras señales, por tanto no se le pueden asignar otro valor inicial.

Sin embargo, para simulación (en los bancos de pruebas) sí puedes dar valores iniciales a las señales, aunque no es necesario. Si en simulación, una señal de tipo `std_logic` no ha recibido valor, mostrará una `'U'` (de *Unknown*, desconocido). En síntesis, no tiene sentido, ya que físicamente la señal tendrá una tensión correspondiente con el cero o el uno lógico, aunque para nosotros su valor sea desconocido o indeterminado.

En nuestro banco de pruebas cambiaremos el valor inicial de la señal de reset a su valor inactivo (`c_off`), en vez del `'0'` que tiene por defecto. El resto de señales las podemos dejar igual.

A continuación, en el subapartado siguiente crearemos el proceso que simula el envío de un dato, posteriormente incluiremos el envío de más datos, y por último añadiremos un modelo de receptor para simulación que nos dirá de manera automática si el envío es correcto y si cumple los tiempos dados por la norma RS-232.

7.5.1. Proceso que modela la orden de enviar un dato

Dentro del banco de pruebas crearemos un proceso llamado `P_Estimulos` que va a modelar la acción de ordenar al transmisor el envío de un dato determinado. En este proceso vamos a hacer lo siguiente:

1. Inicialmente ponemos la señal `transmite` y todos los bits de `dato_tx_in` a cero.
2. Una vez que la señal de reset se haya activado y desactivado, esperamos 70 ns. Esto puedes hacerlo de dos maneras:
   - Calculando el tiempo que transcurre hasta que el reset vuelva a estar inactivo. Del código 5.3 sabes que el reset estará inactivo después de 108 + 75 ns a partir del comienzo. A esto habría que añadirle los 70 ns de la espera que ahora se pide.
   - La manera anterior nos hace depender de los tiempos del proceso del reset, pero es mejor hacerlo depender de los eventos. Es decir, primero esperar a que haya un cambio en el valor del reset (`rst'event`) y que el reset se ponga activo (`rst=c_on`). Posteriormente esperar a que el reset se vuelva a poner inactivo. Y por último, esperar los 70 ns.

   En la figura 7.13 se muestran las dos opciones de manera gráfica. La primera opción, basada en tiempos, se muestra arriba; mientras que la segunda opción, basada en

eventos, se muestra debajo. Nosotros implementaremos la segunda opción. Fíjate que el cronograma del reset se refiere a la placa XUPV2P, pues es activo a nivel bajo.

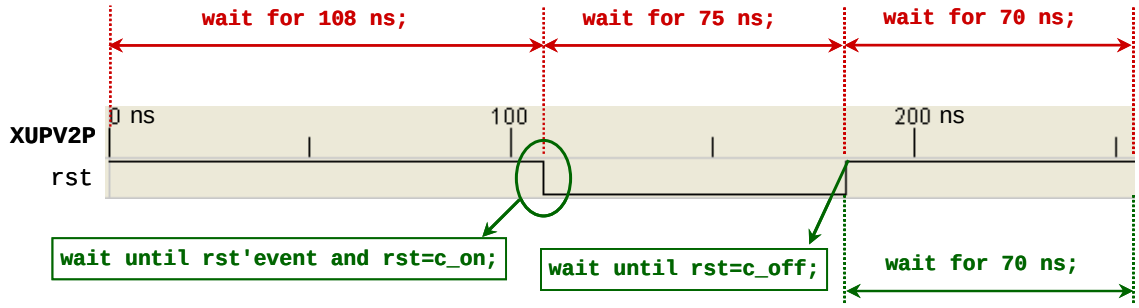


Figura 7.13: Captura de eventos del reset basados en tiempos o en eventos. Referenciado a la XUPV2P

3. Esperamos a que haya un flanco de subida de la señal de reloj. Esto se hace con la sentencia del código 7.16:

```
wait until clk'event and clk='1'
```

Código7.16: Esperar a que haya un flanco de subida del reloj

4. Enviamos el dato "01010101". Para ello tendremos que poner la señal transmite a uno, dando así la orden de transmitir, simultáneamente asignamos a `dato_tx_in` el valor que queremos transmitir.
5. Esperamos al siguiente flanco de subida de reloj.
6. Quitamos la orden de transmitir y ponemos el dato a transmitir todo a cero.
7. Esperamos indefinidamente. Recuerda que se hace con la sentencia `wait;` (código 5.3).

Ahora, implementa este proceso para simular el circuito (en Modelsim o en el simulador del ISE). Antes de simular conviene estimar cuánto tiempo de simulación requerimos para comprobar que nuestro envío se realiza correctamente. Esto además es necesario para comprobar que la simulación es correcta.

Estamos enviando a una frecuencia de 115200 baudios (o lo que hayas puesto en la constante `c_baud` del paquete `UART_PKG`). Esto significa que cada bit se debería de enviar durante algo menos de 8681 ns ([27]). Como enviamos el bit de inicio, los 8 bits del dato y el bit de fin, el tiempo de transmisión será 10 veces mayor: 86810 ns, y a esto le tenemos que sumar el tiempo del reset (figura 7.13) que es algo menos de 300 ns. Por tanto, nuestra simulación durará unos 87110 ns, así que podemos indicar al simulador que simule 88 µs.

En el apartado siguiente vamos a analizar la simulación, se va a explicar con el ISE Simulator, ya que tiene algunas particularidades, pero se puede hacer de manera similar con el Modelsim. En esta explicación se supone que ya sabes los conceptos básicos de la simulación y el manejo de los simuladores. Esto se explicó en el capítulo 5 y sobre todo en el capítulo 8 del manual de ED2 [17mach]. Para cualquier duda consulta estos capítulos ya que es probable que estén explicadas.

El apartado siguiente es muy importante porque de nada nos vale hacer el banco de pruebas si no sabemos interpretar los resultados de la simulación. Las formas de onda de la simulación pueden resultar ininteligibles si no se analizan con cierto detalle y paciencia.

[27] Calculamos la inversa de los baudios y la multiplicamos por 10^9 para pasarlo de segundos a nanosegundos. En realidad nuestra UART no podrá ser tan exacta, sino que el tiempo tendrá que ser una cantidad múltiplo de nuestro periodo de reloj (apartado 7.4.1)

Para el análisis es imprescindible tener claro cómo deben de comportarse las señales. Comparando los resultados de la simulación con los que nosotros esperamos podremos descubrir si hay algún error en nuestro diseño.

7.5.2. Comprobación de la simulación

Las instrucciones básicas para arrancar la simulación con el ISE Simulator están en el capítulo 8 del manual de ED2 [17mach] por lo que no se repetirán aquí. Para simular con Modelsim consulta el capítulo 5.

Como el ISE Simulator tiene algunas particularidades, se explicará la simulación con esta herramienta. Aún así, deberías poder simular con Modelsim sin problemas.

Una vez que pinchas en *Simulate Behavioral Model* (estando seleccionado el módulo del banco de pruebas) aparece la ventana de las formas de onda. Por defecto se simula durante 1000 ns y están incluidas las señales del banco de pruebas, pero no las señales internas del transmisor de la UART. En la figura 7.14 se muestra la ventana de simulación[28].

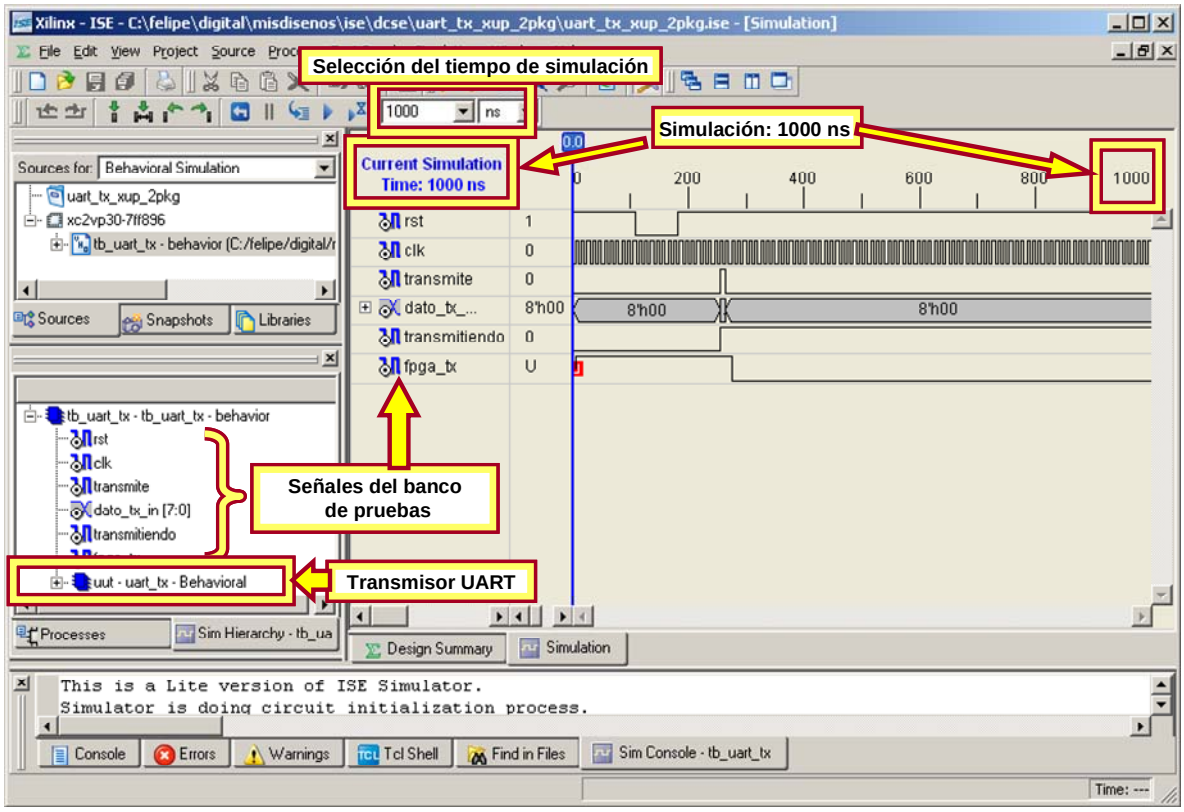


Figura 7.14: Simulación por defecto en el ISE

Si queremos observar todas las señales internas, como son `estado_tx`, `baud`, `cargadato`, `fin_cont8bits`,... (recuerda la figura 7.12), para ello debes de expandir el menú donde se encuentra el transmisor de la UART (figura 7.14), y una vez expandido, seleccionar aquellas señales que queramos que aparezcan. Seleccionaremos todas menos las que ya están en el banco de pruebas (`rst`, `clk`, `transmite`, ...), pinchamos con el botón derecho del

[28] Si has simulado con anterioridad en el mismo proyecto, suele simular el tiempo de la última simulación e incluye las señales que se usaron en esa simulación

ratón y seleccionamos *Add to waveform*. Con esto ya tendremos las señales en el visor de formas de ondas.

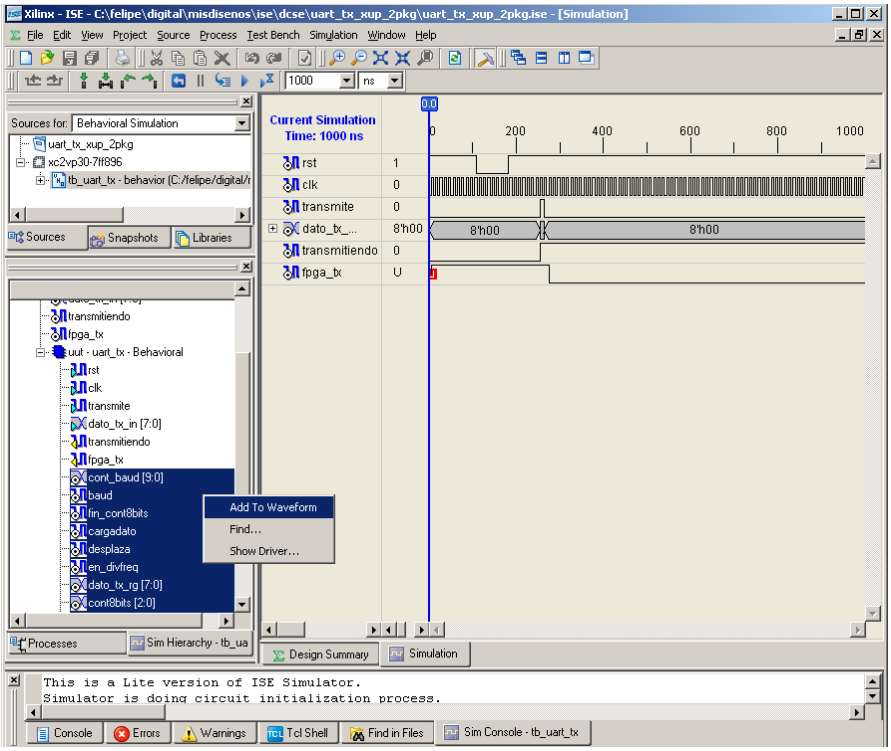


Figura 7.15: Añadir señales en el ISE Simulator

Sin embargo, podemos observar que para las señales que hemos añadido no se muestran sus formas de onda. Así que tendremos que reiniciar la simulación. Para ello, pincha en *Simulation→Restart* (o busca el icono). Y ahora pon 88 µs en la selección del tiempo de simulación (mira la figura 7.14 para ver dónde está). Por último pincha en *Simulation→Run For Specified Time*. Con esto se mostrarán las formas de onda de las señales hasta los 88 µs.

Para ver la simulación completa pincha en *Simulation→Zoom→To Full View* [29]. Ahora es el momento de comprobar si nuestro circuito funciona como debe. La figura 7.16 muestra las formas de onda que aproximadamente deberían aparecer.

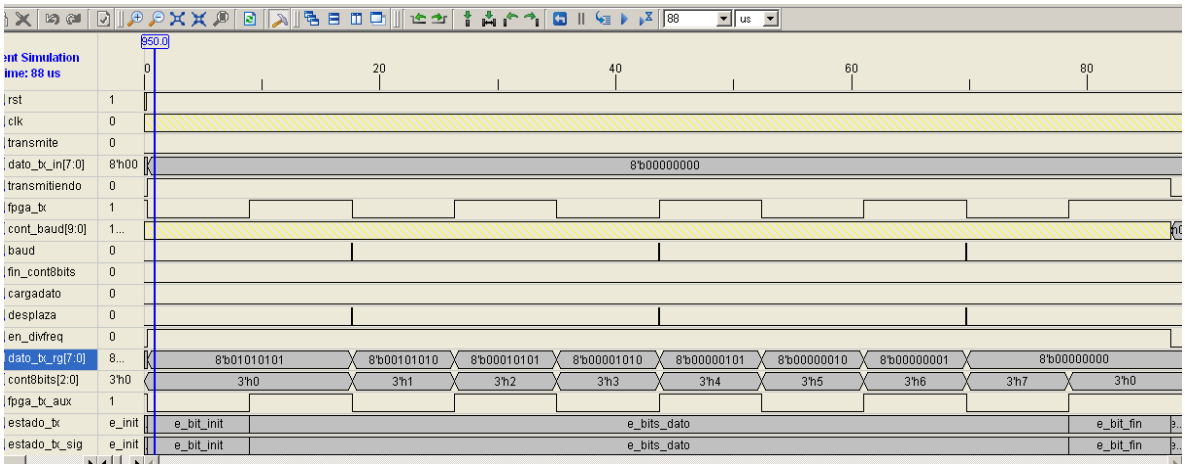


Figura 7.16: Simulación del transmisor con señales internas durante 88 µs

[29] En Modelsim, extrae la ventana de las formas de onda (figura 5.9) y picha en *View→Zoom→Zoom Full*

Ahora pasaremos a comprobar que el circuito funciona bien.

Lo primero que tienes que mirar es la señal `fpga_tx`. Esta señal debe de haber enviado el dato "01010101" conforme al protocolo RS-232. Como ya sabemos las especificaciones del protocolo, el aspecto de la señal debe ser como el mostrado en la figura 7.17. Acuérdate que el primer bit del dato que se envía es el bit 0, por lo tanto, tiene que ser un '1'.

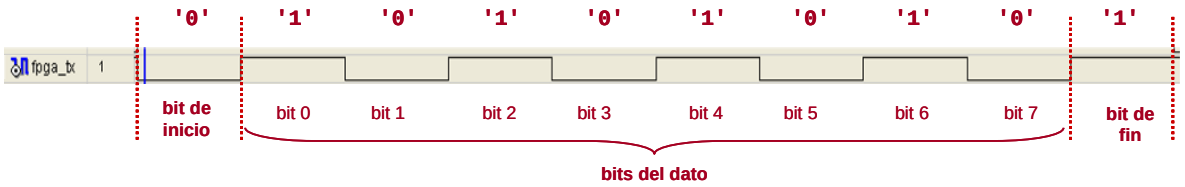


Figura 7.17: Forma de onda de la señal `fpga_tx` después de simular 88 µs

Si el resultado de tu simulación no fuese como el mostrado en la figura 7.17 tendremos que buscar qué ha fallado. De todos modos, no es preocupante, rara vez los diseños funcionan a la primera.

Hay muchas cosas que pueden haber fallado, así que vamos poco a poco y empezaremos por el principio. A continuación se muestran una serie de pasos que debes comprobar:

1. Comprueba que el banco de pruebas hace lo que queríamos que hiciese

   Nuestro banco de pruebas era muy sencillo, así que esta comprobación va a ser fácil. Lo primero es aumentar el zoom de las formas de ondas. Para ello pincha en *Simulation→Zoom→In* (o busca el icono[30]), tendrás que pinchar unas 7 veces o hasta que empieces a ver las transiciones del reloj.

   A continuación llevamos la simulación al inicio arrastrando la barra inferior a la izquierda. La señal `dato_tx_in` la ponemos en formato binario (pinchando encima de ella con el botón derecho).

   Tenemos que comprobar que la señal de reset se comporta como muestra la figura 7.13 (o invertida si trabajamos con la Nexys2). Y que aproximadamente a los 260 ns se activa la señal `transmite`, esta activación debe durar un ciclo de reloj. Simultáneamente la señal `dato_tx_in` debe de cambiar a "01010101". Posteriormente ambas señales se ponen a cero.

   Si esto está bien nos hace pensar que el banco de pruebas es correcto y que el fallo debe de estar en el transmisor. A continuación pasamos a comprobarlo.

2. Comprueba que hay cambio de estado con la orden de transmitir

   A raíz de que se activa la orden de transmitir debe suceder:

   - El estado (`estado_tx`) pasa de `e_init` a `e_bit_init`
   - La señal `carga_dato` se activa en el mismo ciclo de reloj
   - El puerto de salida `transmitiendo` se pone a uno
   - La habilitación del contador (`en_divfreq`) se pone a uno
   - El biestable donde se guarda el dato a transmitir (`dato_tx_rg`) se carga con el valor que queremos enviar (esto sucede en el ciclo de reloj siguiente).
   - La salida del transmisor (`fpga_tx`) se pone a cero (quizá tarde uno o dos ciclos de reloj) indicando que se empieza a enviar el bit de inicio.

[30] En Modelsim: *View→Zoom→Zoom In*

- El contador de los bits que se envían (`cont8bits`) debe mantenerse a cero
- Las señales `baud`, `desplaza`, `fin_cont8bits` deben seguir a cero
- Y por último, el contador del divisor de frecuencia (`cont_baud`) debe estar en cero y a partir del cambio de estado debe de empezar a contar cada ciclo de reloj. Haz que este contador se represente como entero sin signo (*unsigned*) para que sea más fácil de ver la cuenta.

En caso de que algo no funcione, identifica en el código VHDL dónde está el fallo y corrígelo. Cuando lo tengas bien, pasa al siguiente punto. Como puedes ver, se trata de ir comprobando que las especificaciones se cumplen.

3. Comprueba que se envía el bit cero del dato

Si todo lo del punto anterior era correcto, ahora tienes que comprobar que transcurridos alrededor de 8681 ns, el transmisor cambia de estado y pasa a enviar el bit cero del dato. Como se empezó el envío sobre los 260 ns, tendrás que ir al tiempo 8950 ns, para ello pincha en *Simulation→Goto Time* e introduce el tiempo deseado. Debería aparecer algo similar a lo mostrado en la figura 7.18.

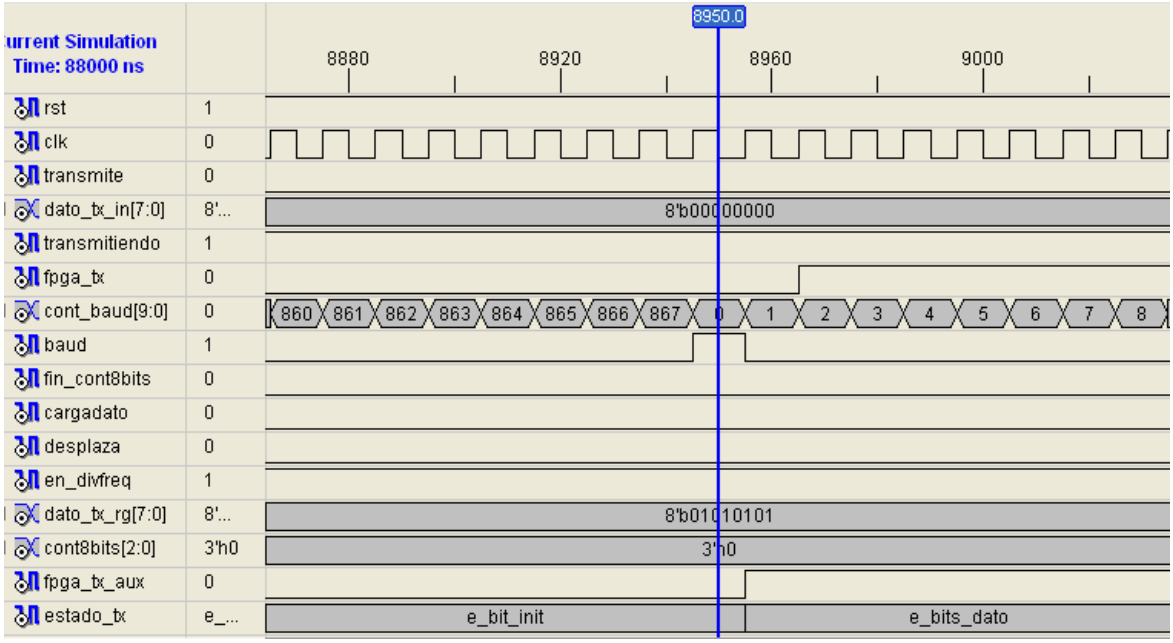


Figura 7.18: Formas de onda para la transición a los bits de datos

Como puedes ver, la señal `baud` se pone a uno, el estado cambia a `e_bits_dato` y el puerto `fpga_tx` empieza a enviar el bit 0 del dato, que es un uno.

Si no ocurre nada de esto, comprueba la señal `cont_baud`. Esta señal debe de incrementar su cuenta en cada ciclo de reloj. Como ya vimos en el apartado 7.4.1, el fin de cuenta depende de la frecuencia de reloj y de los baudios. Para la XUPV2P y a 115200 baudios, el fin de cuenta será 867. Para la Nexys2, como la frecuencia es la mitad, la cuenta también será de la mitad: 433. Esto lo calculaste en la constante `c_fin_cont_baud` (código 7.2).

Entonces comprueba que:

- La señal `cont_baud` realiza la cuenta correctamente y se pone a cero después de su fin de cuenta
- Después del fin de cuenta de `cont_baud` se debe activar la señal `baud`.

- Comprueba que hay cambio de estado desde `e_bit_init` a `e_bits_dato`.

En general comprueba que se cumplen las transiciones de la tabla 7.4.

4. Comprueba que se envían el resto de los bits del dato

   De manera similar al punto anterior, tienes que comprobar que se van enviando el resto de bits correctamente. Tendrás que comprobar que:

   - La señal `cont_baud` realiza la cuenta correctamente y se pone a cero después de su fin de cuenta
   - La señal `desplaza` se activa con la señal `baud`.
   - El registro de desplazamiento `dato_tx_rg` desplaza sus bits a la derecha.
   - La cuenta de los bits enviados `cont8bits` va creciendo con cada nuevo envío de bits

5. Comprueba la transición del último bit de datos al bit de fin

   Esta transición suele dar problemas porque la sincronización de la cuenta con el cambio de estado hay que hacerla con cierto cuidado. En la figura 7.19 se muestra cómo puede ser la transición. Cuando se activa la señal `baud` y la cuenta `cont8bits` va por 7 ocurre la transición al nuevo estado porque la señal `fin_cont8bits` se activa. Fíjate que en la figura 7.19 la señal `baud` y `fin_cont8bits` se activan simultáneamente. No tiene por qué ser así, pero depende de cómo esté hecho el resto del circuito, podría dar algún problema.

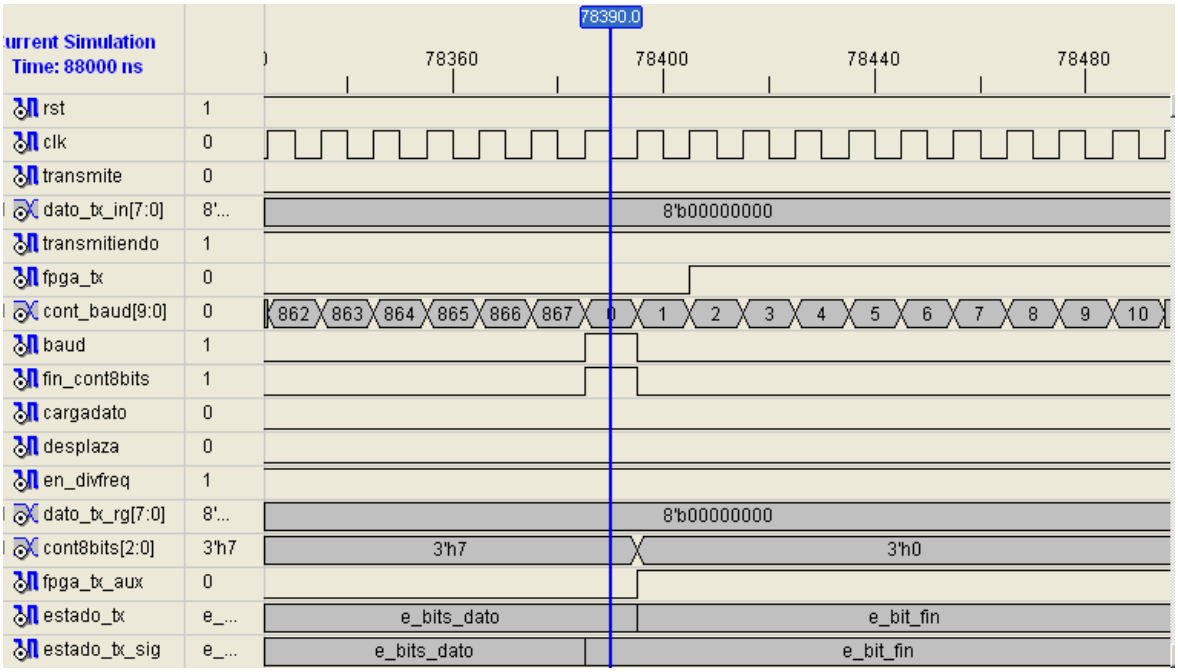


Figura 7.19: Formas de onda para la transición al bit de fin

   A raíz de la activación de la señal `fin_cont8bits` el estado debería pasar a `e_bit_fin`. Con lo que el puerto `fpga_tx` se debe de poner a uno (por ser el bit de fin).

6. Comprueba la transición del bit de fin al estado de reposo:

   Si todo lo anterior te funciona porque has tenido que corregirlo, esta última parte debería serte fácil. Tendrás que comprobar que se activa la señal `baud` después del fin de cuenta. Y la activación de esta señal nos debería de llevar al estado de reposo `e_init`. Debido a esto, la salida `fpga_tx` debe de seguir a uno (reposo) y comprueba que las señales están como las de la tabla 7.4.

7. Comprueba los tiempos:

   Si has llegado hasta aquí, significa que las formas de onda tienen un aspecto similar a las de la figura 7.16 y sobre todo, el dato que se transmite por el cable RS-232 es igual al de la figura 7.17. Sin embargo, puede ser los tiempos no sean exactos, aunque seguramente sí lo sean, pues has hecho comprobaciones que en cierta medida comprueban tiempos.

   Ahora, para comprobar los tiempos, vuelve a mostrar la simulación completa (*View→Zoom→To Full View*[31]). Pon el cursor en 40 μs, que más o menos debe estar en el envío del tercer bit. Para poner el cursor en un tiempo, o bien lo arrastras con el ratón o bien pinchas en *Simulation→Goto Time...* e introduces el tiempo[32].

   Ahora queremos llevar el cursor al momento que ocurre la transición de la señal `fpga_tx` desde el bit 2 hasta el bit 3. Para ello, selecciona la señal `fpga_tx` y pincha en *Simulation→Goto Previous Transition*, y el cursor irá al tiempo (35 μs). En Modelsim tienes que buscar el icono  para ir al transición previa (también existe un icono similar en el ISE Simulator).

   Así que podemos llevar los cursores de transición en transición y ver en qué tiempos ocurren.

   Se pueden insertar varios cursores, en el ISE: *Simulation→Markers→Single Marker*. En Modelsim se insertan: *Insert→Cursor*. El Modelsim indica en la parte inferior la diferencia de tiempos entre dos cursores. Así que si llevas los dos cursores justo en las transiciones de dos bits consecutivos, puedes ver si la diferencia de tiempo se corresponde con la especificada. En nuestro caso muestra 8680 ns (muy cercano a los 8681 ns que habíamos calculado).

   Para calcular diferencias de tiempos en el simulador del ISE pinchamos en *Simulation→Markers→Measure Marker* y pinchamos en donde queramos situarlo. Aparecerán dos cursores, con una diferencia de tiempos encima de ellos. Seleccionamos uno de ellos y lo llevamos a la transición del bit 2 al 3 (no te olvides de seleccionar primero la señal y luego pinchar en *Simulation→Goto Next Transition*). Haz lo mismo con el otro extremo del cursor, llevándolo a la transición del bit 3 al 4. El ISE Simulator nos indica que la diferencia de tiempos es de 8,7 μs. Si hacemos zoom de modo que salga el tiempo en nanosegundos, tendremos la medida exacta. Si haciendo zoom perdemos el cursor, pinchamos en *Markers→Goto Previous Marker* hasta encontrarlo.

   Por último, comprueba que desde el bit de inicio hasta el bit de fin transcurren 86800 ns. También puedes comprobarlo midiendo el tiempo en que `transmitiendo` está a uno.

   Si todas las comprobaciones que hemos hecho son correctas, es probable que el diseño lo tengas bien. Sin embargo, aún no lo hemos comprobado de manera suficiente. Por tanto en los siguientes apartados realizaremos unos bancos de pruebas más exhaustivos. Es importante que aprendas a detectar por ti mismo los errores, para que puedas ser independiente en las prácticas siguientes.

[31] En Modelsim *View→Zoom→Zoom Full*

[32] En Modelsim *View→Goto Time...* cuando introduces 40 μs, lo introduces en nanosegundos (40000)

7.5.3. Proceso que modela las órdenes de enviar varios datos

En el banco de pruebas anterior comprobamos el envío de una trama, sin embargo, muchas veces los errores se producen al enviar una segunda trama, ya que, por ejemplo, puede pasar que no se hayan inicializado bien el contador de los bits de envío.

Por otro lado, en los bancos de pruebas también hay que considerar el funcionamiento erróneo. Por ejemplo, qué pasaría si antes de haber terminado el envío, se ordena un envío nuevo. En principio hemos especificado que cuando se está enviando, el transmisor no va a atender ninguna petición de envío. Por lo tanto, tenemos que comprobar este tipo de cosas.

Así que vamos a crear un nuevo banco de pruebas que envíe cuatro datos y además realice órdenes de envío antes de que el transmisor haya terminado de enviar dato.

Para conservar el banco de pruebas anterior, realiza una copia del fichero del banco de pruebas y llámala `tb_uart_tx_2.vhd`. Edítala y cambia de nombre a la entidad, llamándola `tb_uart_tx_2`, cambia también el nombre de la entidad que aparece en la arquitectura. Debería quedar como indica el código 7.17

```
...
 Entity tb_uart_tx_2 is
 End tb_uart_tx_2;

 Architecture tb ot tb_uart_tx_2 IS
  COMPONENT uart_tx
  ...
```

Código 7.17: Nombre de la nueva entidad para el banco de pruebas

Ahora desde el ISE, selecciona *Behavioral Simulation* (figura 5.2). Pincha en *Add Existing Source* y añade el nuevo fichero que acabamos de crear. De este fichero mantenemos todo casi igual menos el proceso de los estímulos (`P_estimulos`).

Vamos a crear una constante en la que guardaremos los cuatro datos que queremos enviar. Esta constante la crearemos en la misma arquitectura del banco de pruebas. Como la constante es un vector de vectores de 8 bits, tenemos que crear un tipo de datos que será un vector de `std_logic_vector` de 8 bits. En el código 7.18 se muestra la declaración que tienes que incluir.

```
  type vector_slv is array (natural range <>) of std_logic_vector(7 downto 0);
  constant datos_test : vector_slv := ("10001101","01010101","11001010","00101101");
```

Código 7.18: Declaración de un tipo de datos que es un vector de `std_logic_vector` (slv) y la constante con los datos a probar

En la declaración del tipo, el rango "<>" indica que está abierto y se especificará en la declaración de la señal o constante. En la declaración de la constante, el rango viene especificado por el número de datos incluidos en la inicialización (4). En este caso, por ser un rango de tipo natural es ascendente, empezando en el índice cero ("10001101") y terminando en 3 ("00101101").

Hasta el primer envío lo haremos igual que como lo teníamos en el proceso `P_estimulos` (apartado 7.5.1), pero en vez de asignar a `dato_tx_in` directamente un valor constante, le asignaremos el contenido del índice cero de la constante `datos_test`. Para ello, la asignación la debes hacer como se muestra en el código 7.19.

```
 dato_tx_in <= datos_test(0);  -- asignamos el dato del indice cero
```

Código 7.19: Asignación del valor del índice cero de la constante

Ahora vamos a quitar la última sentencia del proceso (`wait;`) y vamos a hacer lo siguiente:

1. Esperamos hasta que termine la transmisión del envío anterior. Esto lo podemos detectar cuando veamos que la señal `transmitiendo` se pone a cero (si no te acuerdas, en la figura 7.13 se explicaba cómo hacer esto).
2. Una vez que hayamos detectado que se ha terminado la transmisión, vamos a empezar el segundo envío, y te recomendamos que pongas un comentario que indique el comienzo del segundo envío. Como ya sabes, los comentarios son muy importantes, no sólo para ti, sino para aquellos que trabajan contigo o te revisan el trabajo y intentan averiguar qué es lo que has querido hacer.
3. Ahora comienzas a realizar el **segundo envío**, ya sabes cómo se hace:
   - Ponemos `transmite` a uno
   - Asignamos a `dato_tx_in` el dato que hay en el índice 1 de la constante `datos_test` (recuerda que empieza en cero, por eso es el índice 1 y no el 2)
   - Esperamos al siguiente flanco de subida de reloj (recuerda el código 7.16)
4. Después de ordenar el segundo envío, podríamos poner la señal `transmite` a cero. Pero no lo vamos a hacer para probar qué ocurre si la señal `transmite` dura más de un ciclo de reloj. Esto sería normal si, por ejemplo, la señal `transmite` estuviese asociada a un pulsador. Entonces lo que haremos será:
   - Ponemos (dejamos) `transmite` a uno. En realidad no haría falta, porque guardaría su valor, pero lo dejamos para darnos cuenta de lo que estamos haciendo.
   - Ponemos todos los bits `dato_tx_in` a cero. Esto lo hacemos para detectar si hay fallo, al cambiar el dato, si en el segundo envío todos los bits son cero, implica que no ha cargado bien el dato.
   - Esperamos 200 ns.
   - Ahora sí, ponemos `transmite` a cero.
   - Dejamos todos los bits de `dato_tx_in` a cero.
   - Y esperamos a que termine la transmisión, es decir detectamos cuando `transmitiendo` sea igual a cero.
5. Todavía no hemos empezado el tercer envío y vamos a cambiar algo las entradas esperando que no ocurra nada. En la simulación habría que comprobar que no ocurre nada por este cambio
   - Mantenemos `transmite` a cero
   - Asignamos a `dato_tx_in` el dato del anterior envío con los bits negados:

```
dato_tx_in <= not datos_test(1);
```

Código 7.20: Negamos los bits del segundo envío

   - Y ahora esperamos un ciclo de reloj, ya que para que las señales actualicen su valor tenemos que poner un `wait`.
6. Empezamos el **tercer envío**:
   - Ponemos un comentario que lo indique
   - Ponemos `transmite` a uno
   - Asignamos a `dato_tx_in` el dato que hay en el índice 2 de la constante `datos_test`
   - Esperamos un ciclo de reloj (esperamos al siguiente flanco de subida)

7. Después de ordenar el tercer envío, ponemos otro dato en `dato_tx_in` para comprobar que no lo envía:
   - Ponemos `transmite` a cero
   - Asignamos a `dato_tx_in` el dato **negado** que hay en el índice 2 de la constante `datos_test`
   - Esperamos 300 ns
   - Ponemos `transmite` a uno (esto no debería cambiar nada)
   - Esperamos 100 ns
   - Ponemos `transmite` a cero
   - Ponemos todo `dato_tx_in` a cero
   - Esperamos 200 ns
   - Y esperamos a que termine la transmisión (detectamos cuando `transmite='0'`).
8. Esperamos la mitad del tiempo en que se envía un bit. Es decir, estamos probando qué pasa si mantenemos la UART inactiva. Este tiempo lo vamos a poner en una constante en el paquete `UART_PKG`, igual que lo hicimos con `c_period_ns_clk` (código 7.14).

```
  constant c_period_ns_baud      : natural   := 10**9/c_baud;
```

Código 7.21: Tiempo que transcurre en enviar un bit

   - Así pues esperamos la mitad del tiempo de un bit, similar a lo que hicimos con la sentencia del reloj (código 7.15) pero con la constante `c_period_ns_baud`.
9. Realizamos el **cuarto envío** (el último). Como la última espera ha sido de tiempos, y no de evento, esperamos un ciclo de reloj para asegurarnos que los valores que ponemos van a durar un ciclo de reloj completo. Para entenderlo, fíjate en la figura 7.20: hemos esperado 48 ns, que coincide con 2 ns antes del flanco de subida del reloj. Si en ese momento ponemos la señal s a uno, y esperamos un ciclo de reloj, la señal s sólo estará activa durante 2 ns, en vez de estar activa durante un ciclo completo de reloj. Especialmente grave es el caso de que esperemos por tiempo y coincida justa en el flanco del reloj: ¿estamos antes o después de la subida?

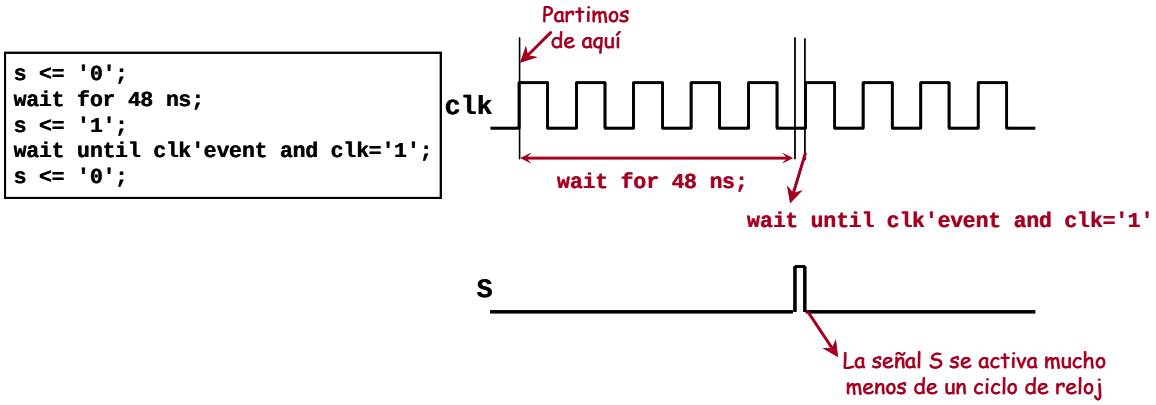


Figura 7.20: Esperar por tiempos y luego por eventos puede producir señales de ancho muy inferior al ciclo de reloj

Para evitar este tipo de situaciones, cuando queramos activar una señal durante un ciclo de ciclos de reloj, será conveniente hacer una espera por eventos de una señal sincronizada con el reloj (no tiene que ser el reloj, puede ser la señal `transmitiendo` como hemos hecho anteriormente). La figura 7.21 explica de manera gráfica la solución.

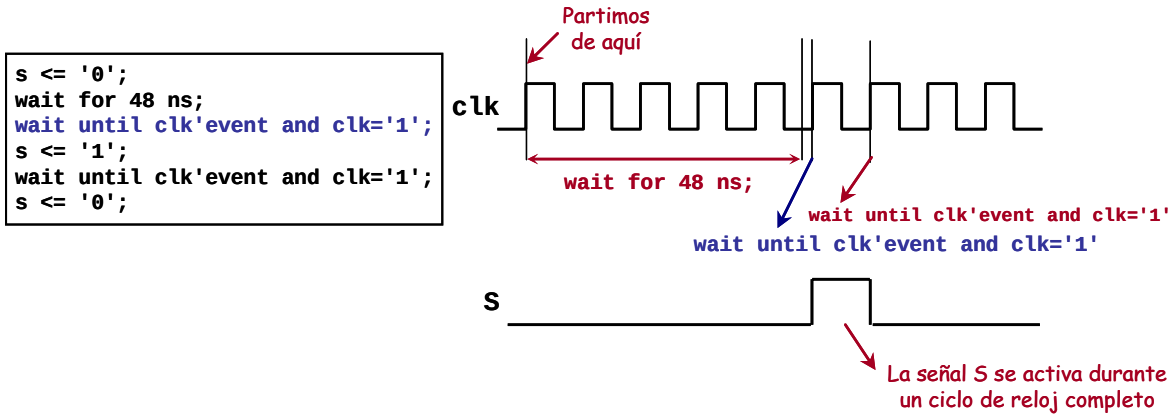


Figura 7.21: Solución a la espera por tiempos seguida de espera por eventos (figura 7.20)

Así que para realizar el cuarto envío haremos:

- Insertamos un comentario indicando que empezamos el cuarto envío.
- Esperamos al siguiente flanco de subida del reloj para sincronizar (figura 7.21).
- Ponemos `transmite` a uno
- Asignamos a `dato_tx_in` el dato que hay en el índice 3 de la constante `datos_test`
- Esperamos al siguiente flanco de subida del reloj

10. Terminamos la transmisión. Antes de esto, declararemos una señal llamada `fin_envio` de tipo `std_logic`, y la inicializaremos a cero. Esta señal nos servirá para saber cuando hemos terminado de ordenar el último envío. Más adelante encontraremos su utilidad
    - Insertamos un comentario
    - Ponemos `transmite` a cero
    - Ponemos todo `dato_tx_in` a cero
    - Ponemos la señal `fin_envio` a uno.
    - Esperamos indefinidamente: `wait;`

Con esto ya tendríamos todo el banco de pruebas. Ahora queda simularlo siguiendo los pasos del apartado 7.5.2. Como son cuatro envíos y vimos que un envío serían unos 88 μs, simularemos 352 μs, o un poco más.

Una vez tengas las formas de onda, tendrás que observar si se aprecian los cuatro envíos, si la máquina de estados cambia a los estados adecuados, los contadores cuentan correctamente...[33] En fin tendrás que realizar los pasos del apartado anterior pero considerando los cuatro envíos. Y antes de nada, observar si los estímulos del banco de pruebas cumplen las especificaciones. Es una tarea de bastante paciencia, pero no es tan complicada.

7.5.4. Autocomprobación en el banco de pruebas

Hemos visto que comprobar un banco de pruebas no es que sea excesivamente complicado (aunque lo pueda llegar a ser) sino que sobre todo es tedioso. Hay que ir transición a transición comprobando que todo esté como suponemos que debe estar y a la vez, comprobando que se cumplen los tiempos especificados. Esto además se agrava por

[33] Ten en cuenta que en el simulador del ISE, cuando estás viendo toda la simulación (no has ampliado con el zoom) a veces no se muestran transiciones de señales que duran poco tiempo (un ciclo de reloj). En el Modelsim si se muestran estas transiciones con una línea.

el hecho de que cada vez que hagamos un cambio en nuestro diseño tendríamos que volver a realizar toda la comprobación. Por esto es muy interesante la idea de crear un banco de pruebas que él mismo se auto-compruebe, avisándonos de cuándo haya un error y por consiguiente evitándonos tener que revisar cada detalle del banco de pruebas.

Evidentemente, hacer un banco que se auto-compruebe es más complicado y requiere más tiempo. Pero el tiempo invertido en hacerlo se ve recompensado por una detección mucho más rápida de los errores.

Para crear un banco de pruebas que se auto-compruebe lo que tenemos que hacer es codificar en el banco de pruebas las comprobaciones que harías a partir de las formas de onda de la simulación (similar a lo que hicimos en el apartado 7.5.2.).

En nuestro caso, para comprobar que el envío del transmisor de la UART es correcto, podríamos poner un modelo de un receptor de la UART de modo que verifiquemos que lo que ha recibido el modelo del receptor se corresponde con lo que ha enviado el transmisor.

Como estamos haciendo un banco de pruebas, el modelo del receptor no tiene que ser sintetizable y puede estar basado en tiempos y eventos, según las especificaciones del envío RS-232 (figura 7.3). De hecho, ni siquiera tiene que estar referenciado a la señal de reloj. Por lo tanto es un modelo bastante más sencillo de hacer.

Con esto estamos viendo que un mismo circuito se puede describir en distintos **niveles de abstracción**. Por ejemplo, el transmisor que hemos hecho está descrito en lo que se llama **nivel de transferencia de registros** (RTL[34]). En este nivel, el funcionamiento del circuito se describe en términos del flujo de las señales (o transferencia) entre los biestables (o registros). En RTL, las transiciones están definidas por la frecuencia del reloj. Hay niveles de abstracción más bajos que el RTL como podría ser la descripción del circuito en el **nivel de puertas** lógicas, e incluso más bajo, como lo sería el **nivel de transistores** (describiendo cada puerta en su transistor correspondiente). **Sintetizar** es el proceso de pasar de un nivel de abstracción más alto a otro inferior, y esto es lo que hace el ISE de manera automática cuando queremos pasar nuestro diseño a la FPGA.

También hay niveles más altos que el RTL, como el **nivel de algoritmo**, en el que se describe el funcionamiento como si se programase en un lenguaje de programación. Este sería el nivel en el que vamos a describir el receptor de la UART para el banco de pruebas. Veremos que en este nivel la descripción es mucho más sencilla que en RTL.

Ahora vamos a empezar a realizar el modelo del receptor que incluiremos en el banco de pruebas. Para ello, de manera similar al apartado 7.5.3, hacemos una copia del banco de pruebas anterior, y al nuevo le llamamos `tb_uart_tx_3.vhd`. Editamos el fichero y cambiamos de nombre a la entidad, llamándola `tb_uart_tx_3` (recuerda código 7.17). Por último, añadimos el fichero al proyecto del ISE.

A continuación, en el banco de pruebas incluimos un nuevo proceso que va a ser nuestro modelo del receptor de alto nivel. Al proceso lo llamaremos `P_Receptor`. El esquema de nuestro nuevo banco de pruebas se muestra en la figura 7.22. En este esquema, los bloques con línea discontinua representan los procesos del banco de pruebas. Puedes observar que el modelo del receptor no genera estímulos como el resto de procesos (reset,

[34] RTL: del inglés *Register Transfer Level*

reloj y entradas), sino que su función es leer la señal enviada por el transmisor para comprobar que es correcta.

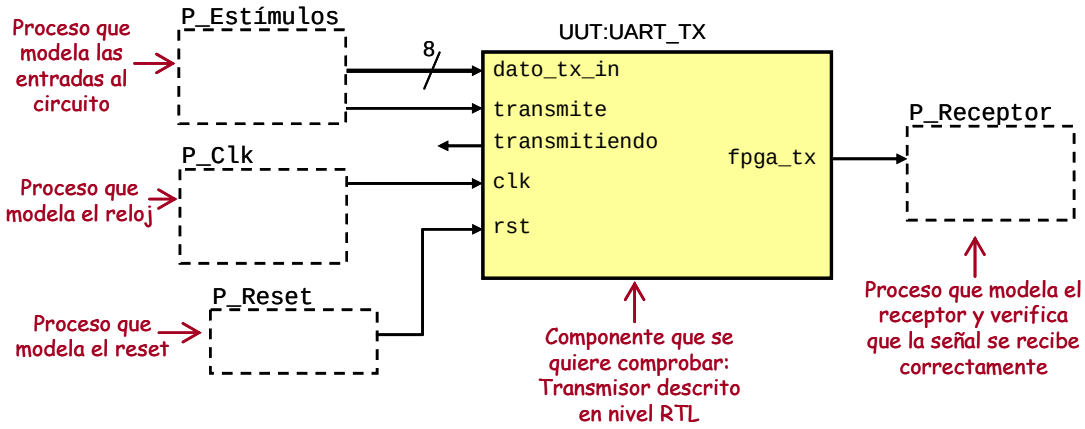


Figura 7.22: Esquema del banco de pruebas del transmisor que incluye un proceso que modela el receptor descrito a alto nivel y verifica que el envío es correcto

Ya que vamos a modelar el receptor, tenemos que tener muy presente cómo es la trama de envío (recuerda la figura 7.3).

Los pasos para comprobar el envío son:

1. Esperamos hasta que el transmisor envíe el bit de inicio. Recuerda que el bit de inicio se indica poniendo la línea de transmisión (`fpga_tx`) a cero. Ya sabes cómo se hace esto:

```
wait until fpga_tx'event and fpga_tx='0';
```

Código 7.22: Espera al flanco de bajada de la señal `fpga_tx`

2. Nos situamos en el tiempo medio del bit de inicio: Una vez que hemos detectado el comienzo de la transición, esperamos la mitad del periodo de envío de cada bit (`c_period_ns_baud/2`) para situarnos justo en medio. Esto es importante, porque si leemos el bit justo en la transición a otro bit, puede pasar que estemos leyendo el bit contiguo. La figura 7.23 muestra de manera esquemática dónde nos sitúa esta espera.

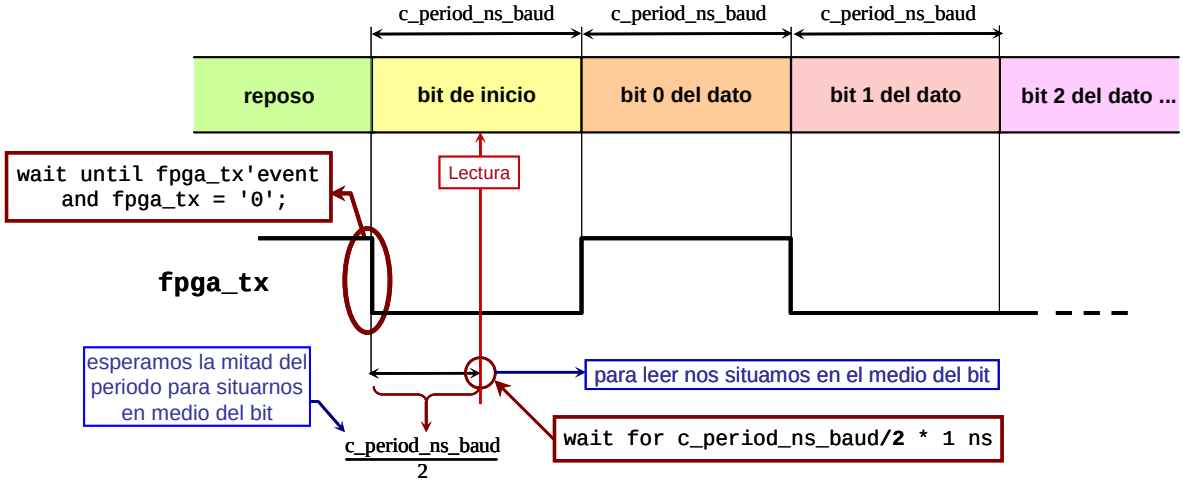


Figura 7.23: Espera para situarnos en medio del bit de inicio

3. Leemos el valor del bit de inicio: Estando en medio del bit de inicio, procedemos a leer su valor. Aunque para el bit de inicio no es necesario leer su valor, porque ya lo sabemos (`'0'`), la lectura la hacemos para comprobar si efectivamente el bit de inicio es cero. En caso de que no sea cero, en VHDL existe la sentencia `assert` que nos permite avisar de errores e

incluso detener la simulación. La sentencia de lectura y verificación de un valor se muestra en el código 7.23.

```
assert fpga_tx = '0'       -- Si no se cumple da el aviso
  report "Fallo en el bit de inicio"
  severity ERROR;          -- niveles de severidad: NOTE,WARNING,ERROR,FAILURE
```

Código 7.23: Ejemplo de una sentencia `assert`

Del código 7.23 podemos observar que la sentencia `assert` tiene tres partes:

- La condición que se quiere verificar (`fpga_tx = '0'`)
- Si **no** se verifica la condición anterior, se muestra el siguiente aviso. Que en el ejemplo es: "*fallo en el bit de inicio*".
- Por último, la gravedad del error. Hay cuatro niveles de error, que de menor a mayor son: `NOTE`, `WARNING`, `ERROR`, `FAILURE`. Dependiendo del nivel de error, se puede ordenar detener la simulación o simplemente avisar[35] .

Importante: date cuenta que estas tres partes forman el `assert`, y que es una única sentencia. Por tanto hay un único punto y coma al final (en medio no hay puntos y comas aunque se pueda separar en varias líneas). Si pusieses un punto y coma después del `assert` implicaría que son dos sentencias, una de ellas un `assert` sin aviso (sin `report`), y la otra sólo tendría `report`, por lo que siempre avisaría, y pensarías que te está avisando de un error.

Otra cosa que puedes observar es que estamos leyendo solamente en el punto medio, como muestra la figura 7.23. Siendo estrictos deberíamos de comprobar que la transmisión se mantiene a cero durante todo el tiempo correspondiente al bit de inicio, y no solo en el punto medio. Sin embargo, para simplificar el banco de pruebas, asumiremos que es suficiente con comprobar el punto medio.

4. Esperamos hasta el punto medio del bit cero del dato. Es decir, esperamos el tiempo correspondiente a un bit (`c_period_ns_baud`). En la figura 7.24 se muestra la espera que estamos haciendo.

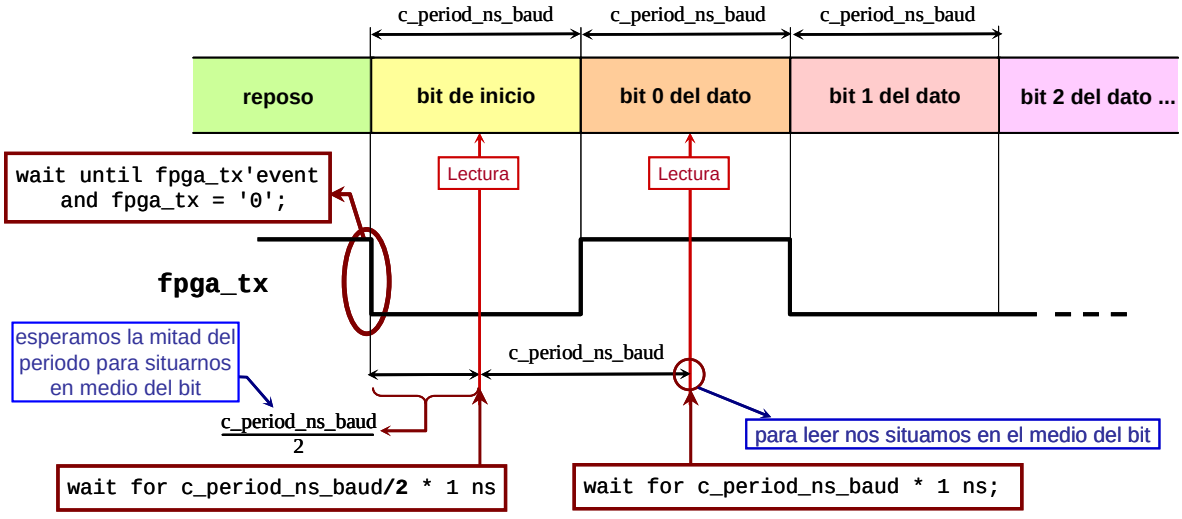


Figura 7.24: Espera para situarnos en medio del bit 0 del dato

[35] En el Modelsim, el nivel donde se detiene la simulación se indica en *Simulate→Runtime Options...* En la ventana que aparece, hay que seleccionar la pestaña *Assertions* y en el apartado *Break on Assertions* se selecciona el nivel deseado

5. Leemos el valor del bit cero del dato. Tenemos que comparar el valor leído con el valor que envió el transmisor. Los datos enviados por el transmisor los tenemos en la constante `datos_test` (recuerda el código 7.18). Ahora estamos en el primer envío (0) y en el bit cero. Por lo tanto la comparación será con `datos_test(0)(0)`. Y el aviso (`report`) tendría que hacer referencia al bit cero del primer envío. La sentencia de lectura y verificación de un valor se muestra en el código 7.24.

```
assert fpga_tx = datos_test(0)(0)    -- Si no se cumple da el aviso
  report "Fallo en el bit cero del dato del primer envio"
  severity ERROR;          -- niveles de severidad: NOTE,WARNING,ERROR,FAILURE
```

Código 7.24: Sentencia `assert` para el bit 0 del primer envío

6. Ahora tendríamos que repetir la espera y repetir la lectura con el `assert` para los restantes 7 bits del dato. Para evitar copiar y pegar siete veces, podemos hacer un bucle `for` como el del código 7.25. Si incluyes el bucle tal como está en el código 7.25, tendrías que borrar lo que hiciste para el bit cero (puntos 4 y 5), pues está incluido en este bucle.

```
for i in 0 to 7 loop
  wait for c_period_ns_baud * 1 ns;        -- punto medio bit i del dato
  assert fpga_tx = datos_test(0)(i)        -- leo y comparo el valor del dato
    report "Fallo en el bit del dato del primer envio (0)"
    severity ERROR;
end loop;
```

Código 7.25: Bucle `for` para comprobar todos los bits del dato

   En VHDL el índice (`i`) de los bucles `for` no hay que declararlo como señal o variable.

   Observa que con el bucle hemos perdido parte de la información del `report` pues ahora no indica en qué bit se ha cometido el fallo. Como en el `report` sólo puede haber texto (`string`), no podemos incluir el índice directamente[36].

7. Una vez que hemos leído el último bit del dato (bit 7), tenemos que hacer una espera más para el bit de fin.
8. Seguidamente leemos el bit de fin, comprobamos si su valor es '`1`' y notificando si hay error.
9. Opcionalmente podemos esperar medio periodo para terminar el bit de fin, pero realmente no es necesario, pues el bit de fin tiene el mismo valor que la línea en reposo.
10. Para terminar el proceso y que no vuelva a ejecutarse más desde el principio, termina con una sentencia `wait;`.

Ahora simula el circuito durante unos 355 μs, como hiciste en el apartado 7.5.3 y observa si hay algún aviso de los `assert`. Estos avisos se muestran en la ventana inferior (mira la figura 7.25). En Modelsim, pinchando en el aviso el cursor se coloca en el instante en que se ha producido. El ISE no tienen esta característica, así que tendrás que mirar el instante de tiempo en el que se produjo y ir a ese tiempo (*Simulation→Goto Time...*). Puede ser que el aviso se haya producido por un fallo en el banco de pruebas que acabas de hacer y no por el propio transmisor que estas probando. Si no encuentras el fallo, revisa que no tengas ningún punto y coma en medio de la sentencia, especialmente entre el `assert` y el `report`.

[36] Esto se puede solucionar creando una función (`a_texto`) que reciba un natural (el índice `i`) y devuelva el `string` correspondiente al número del índice. En el `report` podemos concatenar (`&`), por ejemplo:

```
report "Fallo en el bit " & a_texto(i) & " del primer envio"
```

Se deja como ejercicio opcional incluir esta función

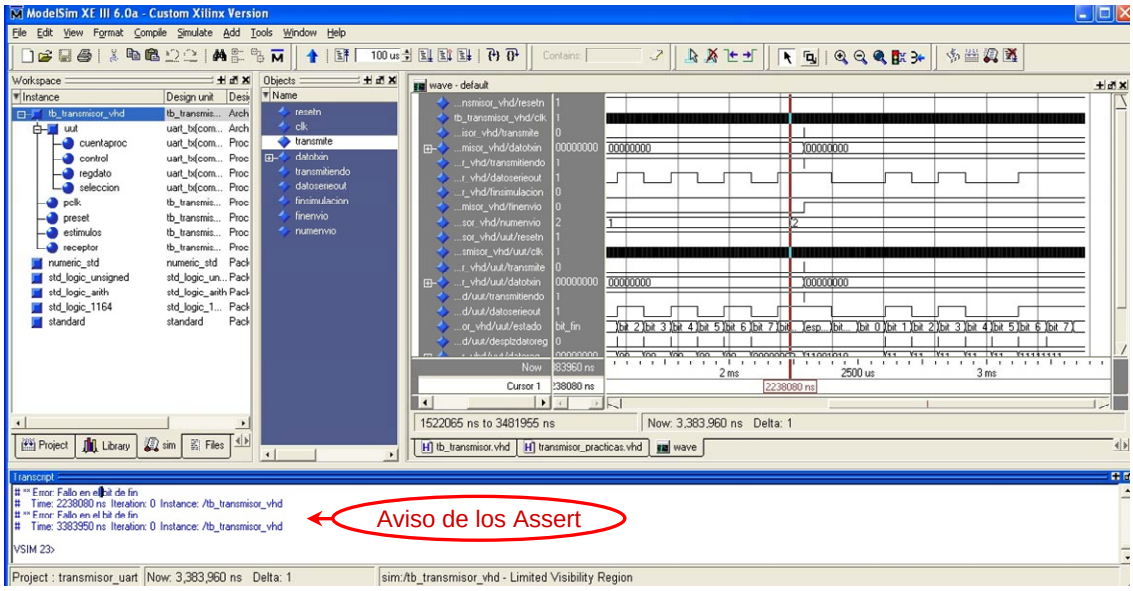


Figura 7.25: Avisos de los `assert` en Modelsim

Quizá te hayas dado cuenta que con lo que hemos hecho hasta aquí sólo estamos comprobando el primer envío, lo que sería el banco de pruebas del apartado 7.5.1, pero ¿qué pasa con los siguientes envíos? ¿cómo los comprobamos?

Podríamos pensar en incluir otro bucle `for` en el proceso que hemos hecho para considerar cada envío. Sin embargo, sabemos que los procesos se ejecutan permanentemente a no ser que terminen en un "`wait;`". Por tanto, quitando el último `wait;` el proceso volvería al principio después del primer envío y realizaría las mismas comprobaciones que acabamos de hacer. Así que no hace falta incluir el bucle `for`, sino quitar el último `wait;` y adaptar el proceso para que haga las comprobaciones adaptadas a cada uno de los envíos.

De manera esquemática, lo que tendrás que hacer es:

1. Quitar el último `wait;` para que el proceso no se detenga después del primer envío.
2. Cambiar la comprobación de los datos enviados. Modificando el `assert` del código 7.25 para que compruebe el dato según el número de envío en el que estamos. Esta modificación se muestra en el código 7.26

```
assert fpga_tx = datos_test(numenvio)(i)
  report "Fallo en el bit del dato"
  severity ERROR;
```

Código 7.26: Modificación del código 7.25 para que compare los bits según el número de envío

   Fíjate que el `report` ya no distingue ni número de envío ni bit. Puedes ampliar lo indicado en la nota al pie número 36 (página 109) para que incluya esta información.

   Observa también que en el código 7.26 se ha creado una variable o señal nueva: `numenvio`. Esta variable va a indicar el número de envío en el que estamos. Así que tendremos que declararla (en el punto siguiente).

3. Declarar de la variable `numenvio`. En los procesos se pueden declarar variables y se declaran antes del `begin`. En simulación se pueden inicializar las variables (en síntesis no se tiene en cuenta la inicialización), por tanto, como estamos haciendo un proceso utilizado simulación, no hay problema. En el código 7.27 se muestra la declaración e inicialización de la variable `numenvio`.

```
P_Receptor: Process
  variable numenvio : natural := 0;
begin
  ....
```

Código 7.27: Declaración de la variable `numenvio` dentro del proceso `P_Receptor`

4. Incrementar el valor de `numenvio` al terminar cada recepción. Esto lo harás al final del proceso. Recuerda que las variables se asignan mediante el operador ":=" y que reciben el valor inmediatamente. A diferencia de las señales que reciben el valor con la sentencia `wait` o al terminar el proceso. En el código 7.28 se muestra la asignación de incremento de la variable `numenvio`, que será la última sentencia del proceso.

```
   ...
   numenvio := numenvio + 1;
end process
```

Código 7.28: Incremento de la variable `numenvio`

Con esto ya comprobaríamos los cuatro envíos. Ahora vuelve a simular el circuito observando si hay nuevos avisos provenientes de los `asserts`. En teoría, si has hecho todos los bancos de pruebas propuestos y has verificado que funcionan bien, podemos pasar a implementarla en la FPGA.

Antes de la implementación, el apartado siguiente te muestra una manera más sofisticada de hacer bancos de pruebas de modo que terminen por sí mismos y no tengas que calcular el tiempo de simulación. Este apartado es opcional, y puedes saltártelo o mirártelo más adelante si estás cansado de los bancos de pruebas.

7.5.5. Bancos de pruebas que terminan por sí mismos ∗

Como muestra el símbolo ∗, este apartado es opcional y muestra cómo hacer que el banco de pruebas termine por sí solo, ahorrándonos tener que calcular el tiempo de simulación que le indicamos al simulador.

El ahorrarnos calcular el tiempo de simulación puede parece un ahorro insignificante, sin embargo, es muy útil para alguien que no ha hecho el banco de pruebas y no tiene la información que nosotros tenemos. Más aún, este cálculo lo hemos hecho a partir de una frecuencia de transmisión de la UART (baudios), que puede variar ya que depende de una constante (`c_baud`). Por lo tanto, el tiempo de simulación es variable, y recalcularlo nos podría implicar tener que leernos la documentación o incluso peor, tener que descifrar el código del banco de pruebas, especialmente si hemos diseñado el banco de pruebas hace tiempo. Como consecuencia, todo esto implica dedicar tiempo que nos lo podemos ahorrar haciendo un banco de pruebas que termine por sí mismo.

Para detener la simulación y por tanto, hacer un banco de pruebas que termine solo, tenemos que detener para siempre todos los procesos que generan estímulos, esto es, los procesos del banco de pruebas.

Vamos a verlo con un ejemplo. Copiamos el fichero del último banco de pruebas que hemos hecho (el fichero `tb_uart_tx_3.vhd` del apartado 7.5.4) y lo llamamos `tb_uart_tx_4.vhd`. Cambiamos también el nombre de la entidad y arquitectura y lo añadimos al proyecto del ISE.

Observamos el banco de pruebas y vemos que de los cuatro procesos, hay dos que terminan con la sentencia "`wait;`", que son el proceso de los estímulos y el proceso del

reset. Por tanto, estos dos procesos terminan por sí mismos, y no tenemos que modificarlos.

De los otros dos procesos, tenemos el proceso del reloj que es indefinido y siempre estará generando la señal de reloj, y el proceso del receptor, que siempre está a la espera de recibir una nueva transmisión.

Tenemos que cambiar estos dos procesos de modo que también terminen con la sentencia "`wait;`" cuando se haya completado la transmisión y recepción.

En el proceso que genera los estímulos habíamos asignado a uno de la señal `fin_envio` (recuerda el punto 10 de la página 105) justo antes de terminar (antes de la sentencia "`wait;`"). Hasta ahora esta señal no la hemos utilizado, y nos servirá para avisarnos sobre cuándo se da última orden de transmisión.

Sin embargo, no debemos de detener la simulación cuando la señal `fin_envio` se pone a uno, pues esta señal indica cuándo se ha dado la última orden de envío. Habrá que esperar a que el transmisor envíe cada uno de los bits con el protocolo RS-232 y que el receptor los reciba.

Ahora tendremos que modificar el modelo del receptor de modo que una vez que la señal `fin_envio` valga uno, y se haya completado la recepción, ponga una nueva señal a uno indicando que se debe terminar la simulación. Esta nueva señal la llamaremos `fin_simulacion`, y la declararemos con valor inicial `'0'`.

Con esta modificación, el proceso del modelo del receptor terminará como muestra el código 7.29.

```
    ...
   if fin_envio = '1' then
      fin_simulacion <= '1';
      -- esperamos un poco para no detener simulacion de repente
      wait for c_period_ns_baud * 1 ns;
      wait;  -- detenemos este proceso despues derecibir el ultimo envio
    end if;
  end process;
```

Código 7.29: Modificación del proceso del receptor del banco de pruebas para que se detenga en la última recepción y genere la señal de aviso del fin de la simulación

Observa que en el código 7.29, con el último "`wait;`" detiene el proceso en caso de que se esté recibiendo el último envío. Observa también que antes del último `wait` se ha incluido una espera de un periodo de la UART para que la simulación no termine repentinamente justo después de la recepción del bit de fin. Dejando así tiempo para que el receptor pase al estado de reposo.

Ahora sólo nos queda detener el proceso del reloj y lo detendremos con la señal `fin_simulacion`. Modificamos el proceso del código 7.15 para que termine como muestra el código 7.30

```
  ...
    if fin_simulacion = '1' then
      wait;  -- detenemos el proceso cuando se haya recibido el ultimo
    end if;
  end process;
```

Código 7.30: Modificación del proceso del reloj para detener la simulación

Con esto ya no tenemos que preocuparnos por definir un tiempo de simulación. En el ISE pincharemos en *Simulation→Run All*; mientras que en Modelsim pincharemos en *Simulate→Run→Run -All*, o en los iconos correspondientes. Con esta orden y debido a que

los procesos terminan con "`wait;`" la simulación se detiene sola y no tendremos que preocuparnos de cuánto tiempo debemos simular. No obstante, siempre puede suceder que te hayas equivocado al hacer estas modificaciones, así que al hacer un cambio debes de comprobar que todo sigue bien. En este caso debes de comprobar que se hacen los cuatro envíos correctamente y que tanto el transmisor como el receptor terminan en el estado inicial.

7.6. Implementación en la FPGA: diseño estructural

Que el circuito sea funcionalmente correcto no implica que ya no vayamos a tener ningún problema en su implementación. Puede ocurrir que tengamos restricciones temporales, de área o de consumo que nos obliguen a realizar modificaciones. Además, a veces ocurre que los resultados de simulación no se corresponden con los de síntesis, ya que, entre otros motivos, por ser el conjunto de síntesis más reducido que el de simulación, la síntesis no siempre traduce fielmente el modelo VHDL.

Ya hemos diseñado el transmisor, que lo hemos simulado y al menos en tería funciona bien. Sin embargo sería inútil implementarlo en la FPGA sin nada más, pues el transmisor necesita una orden para transmitir. Así que tenemos que hacer un circuito sencillo que nos dé algo que transmitir. Para ello emplearemos los pulsadores, pues ya hemos probado que funcionan, y haremos que al pulsar se envíe un byte por el transmisor a la computadora.

Este ejemplo nos servirá para introducir el diseño estructural.

Para comenzar la implementación, volvemos al proyecto en el ISE, allí tendremos el componente transmisor y el banco de pruebas. En la subventana de fuentes (*Sources*) seleccionamos las fuentes para síntesis e implementación (*Sources for: Synthesis/Implementation*).

Ahora crearemos un diseño estructural, donde tengamos el transmisor junto con un interfaz con los pulsadores. El esquema del diseño se muestra en la figura 7.26.

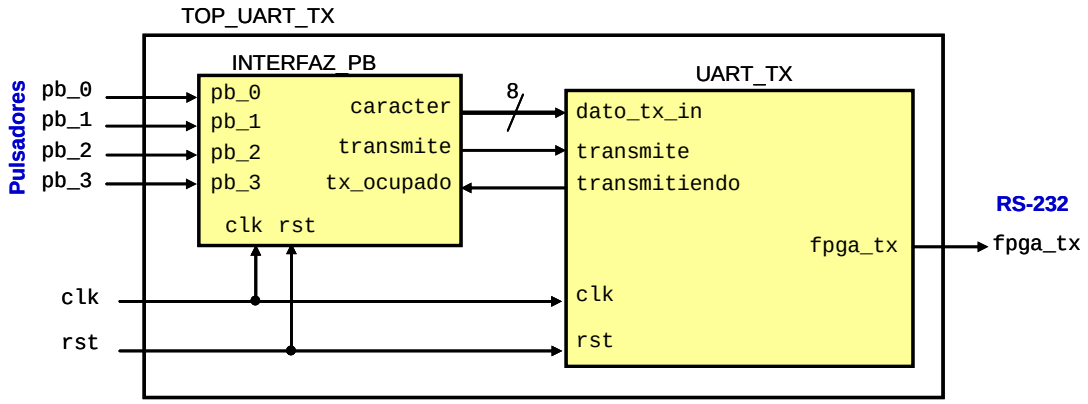


Figura 7.26: Esquema estructural del transmisor de la UART que implementaremos en la FPGA

Cada uno de los bloques de la figura 7.26 representa una entidad VHDL y lo crearemos en un fichero distinto:

- `UART_TX`: es el transmisor de la UART que hemos creado y simulado
- `INTERFAZ_PB`: es el interfaz con los pulsadores que cuando se presionen uno de los pulsadores generará la orden de transmitir un dato concreto a la UART

- `TOP_UART_TX`: es un bloque estructural que va definir las entradas y salidas, y establecer las conexiones entre los dos módulos anteriores.

A continuación se dan guías para diseñar estos módulos

7.6.1. Diseño del interfaz con los pulsadores

Empezamos creando una nueva fuente (*New Source*) de tipo *VHDL Module*, cuyos puertos son los mostrados en la figura 7.26. Todos los puertos serán de tipo `std_logic`, menos `caracter` que será un `std_logic_vector` de ocho bits.

Las especificaciones del módulo `INTERFAZ_PB` son:

1. Recibirá las entradas de cuatro pulsadores. Ya sabemos que estas entradas son totalmente asíncronas y de duración variable.
2. Según el pulsador que se presione, debe de dar la orden al transmisor de enviar un dato determinado. En la tabla 7.5 se muestra la correspondencia de los puertos con los pulsadores de la placa, el dato que deben enviar (en hexadecimal) y el carácter ASCII con el que se corresponde.

Puerto	Pulsadores a los que está conectado		Dato	Carácter ASCII
	XUPV2P	Nexys2		
pb_0	DOWN	BTN0	x"61"	'a'
pb_1	RIGHT	BTN1	x"6C"	'l'
pb_2	UP	BTN2	x"6F"	'o'
pb_3	LEFT	BTN3	x"68"	'h'

Tabla 7.5: Puertos de `INTERFAZ_PB` y su correspondencia con los pulsadores de las tarjetas y el dato que tienen que enviar al transmisor

3. La orden de transmisión debe de durar un único ciclo de reloj y debe enviarse una única orden de envío por cada pulsación.
4. Si se presiona el pulsador mientras el transmisor está enviando un dato (`transmitiendo='1'`), se ignorará la pulsación y no se enviará ninguna orden al transmisor.
5. Es improbable que se presionen dos pulsadores en el mismo instante, en es caso puedes elegir qué envío tendrá prioridad.

Con estas especificaciones intenta realizar el interfaz por ti mismo. Posteriormente, para comparar tu solución o si ves que no te sale, lee las indicaciones que se dan a continuación.

Como la orden de envío tiene que durar un único ciclo de reloj y los pulsadores son asíncronos, haciendo que las señales estén activas durante muchos ciclos de reloj, tendremos que realizar detectores de flanco[37] para cada señal proveniente del pulsador. Incluso, para la placa Nexys2 se debería de hacer un circuito anti-rebotes[38]. Por tanto internamente el circuito podría tener dos procesos:

- `P_detecta_flanco`: Proceso detector de flancos para cada pulsador
- `P_envia_orden`: Proceso que da la orden de enviar los datos determinados por la tabla 7.5

El esquema interno del circuito se muestra en la figura 7.27.

[37] Recuerda el código 2.16. Los detectores de flanco se explican con detalle en el capítulo 5.3.2 del libro de ED2

[38] Práctica 11 del manual de ED2 [17mach]

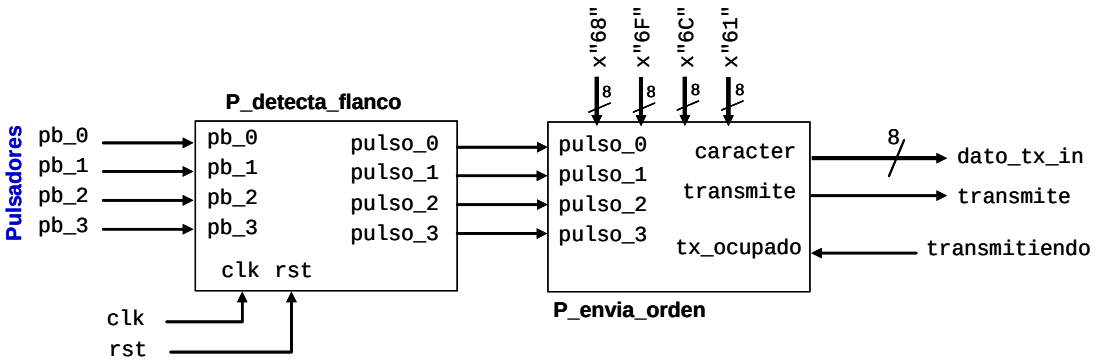


Figura 7.27: Esquema interno de interfaz con los pulsadores

En el proceso `P_detecta_flanco` debes de tener en cuenta que según la placa que uses, los pulsadores serán activos a nivel alto o bajo. Esto ya lo solucionamos con las constantes `c_on` y `c_off` (recuerda el código 6.5), en el reset debes de inicializar los biestables de los detectores de flanco al nivel inactivo (`c_off`) porque de lo contrario, seguramente tengas una orden de envío con el reset. La señales de los pulsos detectados (`pulso_0`, `pulso_1`, ...) las puedes poner activas a nivel alto.

El proceso `P_envia_orden` puede ser combinacional, simplemente tiene que dar la orden de transmitir (`transmite='1'`) cuando alguno de los `pulsos` (flancos detectados) estén activos y la señal `tx_ocupado` no esté activa. En ese caso, se enviará el dato correspondiente al pulsador presionado.

Como es un proceso combinacional, no te olvides de asignar las señales en todos los casos o bien poner una asignación por defecto. Con esto evitas que se formen *latchs* (recuerda el apartado 2.5).

Como último apunte, para asignar un valor hexadecimal, en VHDL se indica con una "x" delante. Por ejemplo:

```
caracter <= x"68";   -- "01101000", la x indica que es hexadecimal; en ascci: h
```

Código 7.31: Asignación de un hexadecimal a un vector de 8 bits

Una vez que realicemos los dos procesos comprobamos la sintaxis en el *ISE* (*Check Syntax*). Fíjate que si tienes seleccionado el módulo `INTERFAZ_PB`, en la subventana *Processes* hay menos opciones. Mientras que seleccionando la `UART_TX` aparecen todas las opciones de implementación. Esto se debe a que la `UART_TX` figura como el módulo de más alto nivel, esto el ISE lo indica con un simbolito con cuadraditos a su izquierda  . Para poner un módulo como el de más alto nivel se pincha con el botón derecho encima de él y se selecciona *Set as Top Module* (figura 7.28). De todos modos, ninguno de estos dos módulos va a ser el superior en la jerarquía, sino que será el `TOP_UART_TX`, que crearemos más adelante.

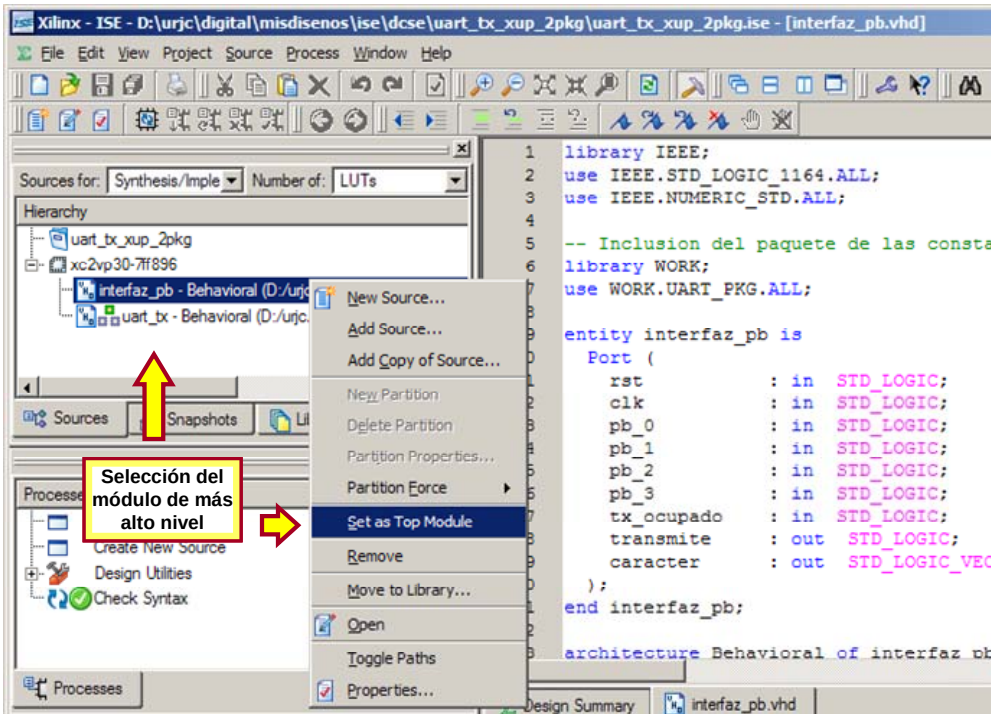


Figura 7.28: Cómo indicar el módulo de mayor nivel

7.6.2. Banco de pruebas del interfaz con los pulsadores

Aunque el interfaz con los pulsadores es un diseño sencillo, es habitual cometer algún fallo, por lo tanto se recomienda hacer un banco de pruebas para asegurar que está bien. Comparado con el banco de pruebas del transmisor (apartado 7.5), este banco de pruebas es mucho más sencillo y no debería suponer ninguna dificultad.

Te recomendamos que lo intentes hacer por ti mismo y luego veas las indicaciones que se dan a continuación.

El interfaz con los pulsadores tiene cinco entradas además del reloj y el reset. De estas entradas, cuatro corresponden a los pulsadores y la otra indica si el transmisor está ocupado. Un banco de pruebas sencillo para este circuito consistiría en modelar el funcionamiento de los pulsadores. Aunque en vez de hacer que los pulsadores estén activos durante milisegundos, para reducir los tiempos de simulación, haremos que estén activos durante unos cientos de nanosegundos (o a lo más microsegundos).

Al activar un pulsador, tenemos que comprobar que la salida `transmite` se active durante un único ciclo de reloj y que, en el mismo ciclo de reloj, la salida `caracter` tenga el valor correspondiente al pulsador. Si te das cuenta, estamos repitiendo las especificaciones del apartado anterior. Así que lo que hay que hacer es modelar el comportamiento de las entradas y comprobar que se cumplen las especificaciones.

En un principio, para simplificar el banco de pruebas, puedes dejar la entrada `transmite` permanentemente a cero. Aunque para ser más rigurosos, podrías implementar un pequeño proceso que modelase de forma simplificada el transmisor. De modo que simplemente active la señal `transmite` durante unos cien ciclos de reloj[39] cada vez que el interfaz con los pulsadores dé la orden de transmitir (`transmite='1'`).

[39] Debería estar activa lo que dura una transmisión, pero lo puedes dejar en cien ciclos de reloj para simplificar y reducir el tiempo de simulación.

Asimismo podrías probar qué sucede si se activan dos pulsadores simultáneamente y también si se activan en tiempos muy próximos.

Al hacer el banco de pruebas, usa las constantes `c_on` y `c_off` para independizarlo de las características de la placa.

Una vez que has comprobado que funciona, puedes pasar al siguiente apartado para implementar en la placa el diseño completo: el transmisor de la UART junto con el interfaz con los pulsadores.

7.6.3. Diseño estructural: unidad de más alto nivel

En este apartado crearemos el módulo que engloba a los anteriores, lo llamaremos `TOP_UART_TX` y la arquitectura será de tipo `Estructural`. Al ponerle este nombre indicamos que la arquitectura no describe un comportamiento sino que indica los bloques (módulos) con los que está formada y las conexiones entre los bloques y los puertos de la entidad.

Los pasos para realizar el diseño estructural son[40]:

1. **Declarar los puertos de la entidad**. Los puertos son los mostrados en la 7.26, todos ellos son de tipo `std_logic`. Recuerda que al declarar la entidad y los puertos estamos indicando que nuestro circuito es como una *caja negra* que se comunica con el exterior con los puertos de entrada y salida. En nuestro caso, esta caja negra se representaría como muestra la figura 7.29.

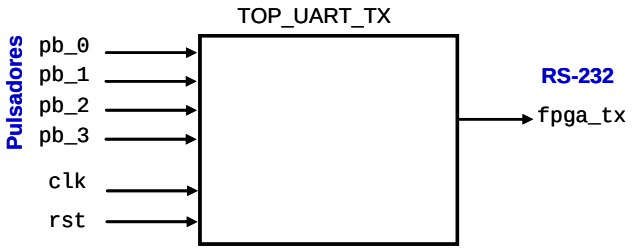


Figura 7.29: Representación esquemática de la entidad de más alto nivel

2. **Nombrar a la arquitectura `Estructural`** (o en inglés `Structural`) de la entidad `TOP_UART_TX`.
3. **Declarar los componentes** que vamos a utilizar. Como es un diseño estructural que está formado por componentes, tenemos que declarar los componentes en la parte declarativa de la arquitectura. Esto ya lo hemos hecho en los bancos de pruebas (recuerda la figura 5.2). Sin embargo, al crear un banco de pruebas, la herramienta ISE declara el componente automáticamente, así que es probable que no te hayas dado cuenta de esta declaración.

   La declaración de un componente es muy similar a la declaración de la entidad. En el código 7.32 se muestra la declaración de la entidad del `INTERFAZ_PB` y en el código 7.33 se muestra la declaración del componente `INTERFAZ_PB`. Las diferencias se han resaltado en rojo.

[40] Si no aparece el componente que acabas de añadir puede ser porque el ISE lo ha añadido como una fuente de simulación. Para que aparezca en síntesis, en *Sources for* selecciona *Behavioral Simulation* (figura 5.2). Allí probablemente esté el componente `TOP_UART_TX`. Pincha en él con el botón derecho y selecciona *Properties*. En *Association* selecciona *Synthesis/Imp + Simulation* y pincha en *Ok*. De todos modos, al principio, mientras no incluyas ninguna sentencia sintetizable, el ISE va a interpretar que es para simulación y no cambiará las propiedades. Así que al principio trabaja desde *Behavioral Simulation* hasta que tengas avanzado el circuito, en ese momento compruebas la sintaxis (*Check Syntax*), le cambias las propiedades y vuelves a trabajar desde *Sources for* *Synthesis/Implementation*.

```
entity interfaz_pb is
 Port (
  rst         : in  std_logic;
  clk         : in  std_logic;
  pb_0        : in  std_logic;
  pb_1        : in  std_logic;
  pb_2        : in  std_logic;
  pb_3        : in  std_logic;
  tx_ocupado  : in  std_logic;
  transmite   : out std_logic;
  caracter    : out std_logic_vector (7 downto 0)
 );
end interfaz_pb;
```

Código 7.32: Declaración de entidad y diferencias con la declaración de componente

```
component interfaz_pb is
 Port (
  rst         : in  std_logic;
  clk         : in  std_logic;
  pb_0        : in  std_logic;
  pb_1        : in  std_logic;
  pb_2        : in  std_logic;
  pb_3        : in  std_logic;
  tx_ocupado  : in  std_logic;
  transmite   : out std_logic;
  caracter    : out std_logic_vector (7 downto 0)
 );
end component;
```

Código 7.33: Declaración de componente[41] y diferencias con la declaración de entidad

Así que incluye las declaraciones de los componentes `INTERFAZ_PB` y `UART_TX` en la parte declarativa de la arquitectura (antes del `begin`).

4. **Referenciar o "*instanciar*" los componentes**. Esto sería como colocar y conectar en nuestro circuito estructural los módulos que vamos a utilizar. En la referencia se indican las conexiones de señales, el llamado *port map*, donde los puertos del componente referenciado se conectan con las señales y puertos del circuito estructural.

   En la figura 7.30 hemos detallado el circuito original (figura 7.26), donde se han diferenciado los puertos de la entidad `TOP_UART_TX` en azul, las señales internas de la arquitectura `TOP_UART_TX` en rojo y los puertos del componente `UART_TX` en verde.

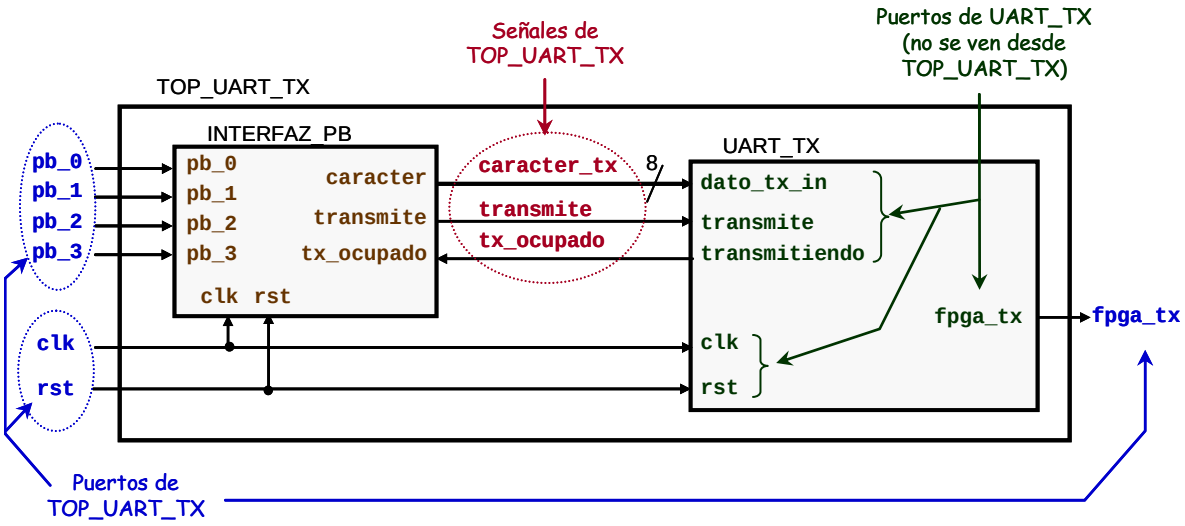


Figura 7.30: Representación esquemática de la arquitectura estructural, con los componentes referenciados y sus conexiones

Los puertos de los componentes "no se ven" desde la arquitectura estructural. Por ejemplo, el puerto de salida `transmitiendo` del componente `UART_TX` no se puede usar directamente por ser un puerto del componente. Esto es lógico, ya que dos componentes podrían tener puertos con el mismo nombre (por ejemplo imagina que hubiese otro tipo de transmisor con la misma señal `transmitiendo`), en ese caso, si pudiésemos hacer referencia a los puertos de los componentes nos crearía un conflicto porque no sabríamos a cuál de los dos puertos nos estaríamos refiriendo.

[41] El VHDL a veces es flexible en su codificación. Por ejemplo, la palabra "`is`" que va después del nombre del componente (en la primera línea) es opcional. Otro ejemplo es que después del "`end component`" se puede incluir el nombre del componente (antes del punto y coma).

Así que los puertos de los componentes los conectaremos con señales o puertos del diseño estructural, y lo que usaremos en la arquitectura serán esas señales o puertos. Estas señales y puertos del diseño estructural no tienen por qué llamarse igual que los del componente, aunque puedan llamarse igual.

En la figura 7.31 se explica cómo se referencia un componente en VHDL. En el punto 1 se pone el nombre de la referencia. Esto es así porque un mismo componente se puede utilizar más de una vez. En el punto 2 se pone el nombre del componente utilizado, que es el nombre de la entidad. En el *port map* se conectan los puertos del componente con los puertos y señales del estructural. A la izquierda los puertos del componente (punto 3) y a la derecha los puertos y señales del estructural (punto 4). En el ejemplo, de estos puertos y señales, tres son señales (punto 5) y el resto son puertos (punto 6). Observa que el nombre de los puertos del componente no tiene que coincidir con los puertos y señales del estructural.

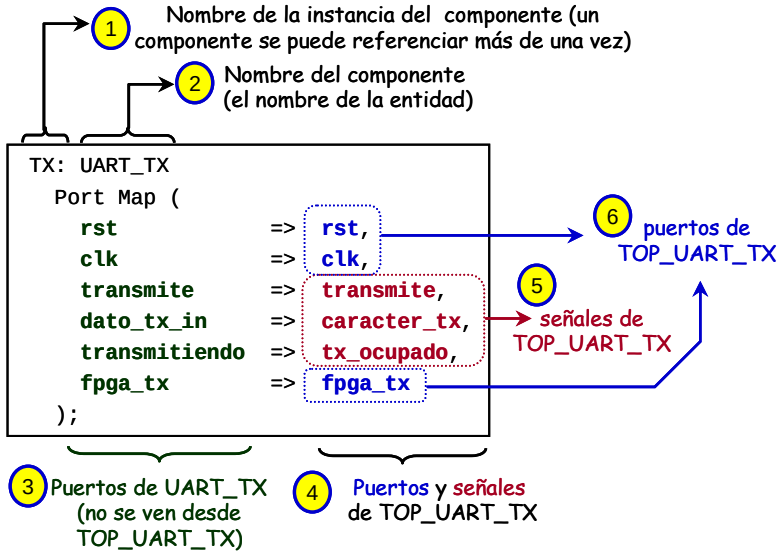


Figura 7.31: Explicación de la referencia o instancia de un componente en VHDL

Ahora incluye la referencia de la `UART_TX` (figura 7.31). A partir de la explicación que se ha dado y observando las conexiones de la figura 7.30, incluye también la referencia del `INTERFAZ_PB`.

5. **Declarar señales**. Observa en las figuras 7.30 y 7.31 las señales del estructural: `transmite`, `caracter_tx` y `tx_ocupado`. Estas señales las tendremos que declarar en la arquitectura. Así pues decláralas en la parte estructural de la arquitectura. Estas señales tendrán que coincidir en tipo y número de bits con los puertos con los que se conectan. De lo contrario aparecerá un error al compilar.

Una vez hayas hecho todos estos pasos, ya puedes comprobar si la sintaxis es correcta. Si aún estás en *Sources for:* ***Behavioral Simulation*** mira la nota al pie 40 de la página 117 para cambiar a síntesis.

Fíjate quién es ahora la unidad de más alto nivel, debería de ser `TOP_UART_TX`.

Ahora crea el fichero `.ucf` para asociar los puertos con los pines. Los pulsadores asócialos según la tabla 7.5. Para la XUPV2P pon el reset en el pulsador del medio, y en la Nexys2 pon el reset en el primer interruptor.

Sintetiza e implementa el circuito en la placa. Conecta el puerto serie del ordenador con la placa.

Y ahora comprobaremos que funciona bien. Para ello abrimos un *hiperterminal*, Windows suele tener uno en *Inicio→ Todos los programas→ Accesorios→ Comunicaciones→ HyperTerminal*. También hay otros de libre distribución como el *RealTerm* [24realt] que es gratuito y tiene opciones muy interesantes para depurar en caso de que algo no vaya bien.

Usaremos el *HyperTerminal* de Microsoft, una vez que se sabe utilizar uno no es difícil usar otro.

Al abrirlo nos aparecerá una ventana para establecer una nueva conexión (figura 7.32), nos pide nombre de la conexión y un icono para ella (esto del nombre y del icono no siempre lo piden en otros *hiperterminales*). Si no nos lo pidiese pinchamos en *Archivo→Nueva Conexión*.

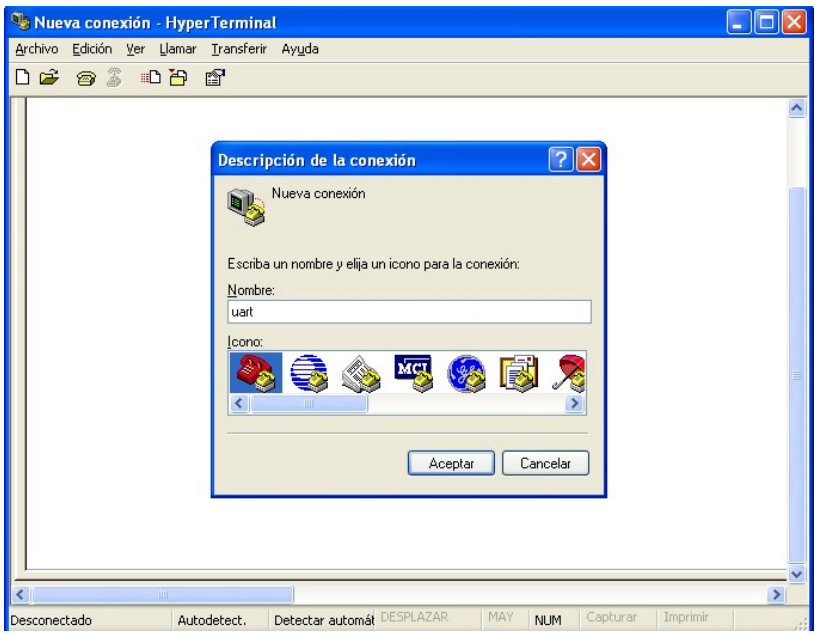


Figura 7.32: Nueva conexión del hiperterminal

Posteriormente (figura 7.33) aparecerá una ventana que dice *Conectar a*, le indicamos *Conectar usando COM1* (u otro COM según tu ordenador).

Y por último nos saldrán las propiedades de la comunicación con el puerto serie (figura 7.34). Después del diseño que hemos hecho, ya estaremos familiarizados con el significado de estas propiedades. Las rellenamos según las características de nuestro diseño: bits por segundo (115200), paridad (sin paridad), bits de parada (1), control de flujo (ninguno), ...

Después de darle a aceptar ya tendremos la conexión, y probamos a presionar uno de los cuatro botones (no el del medio que es el reset para la XUPV2P y no hará nada visible).

Comprueba que en el *hiperterminal* aparecen los caracteres que has enviado. Si te salen, enhorabuena, has logrado implementar un diseño de relativamente complejo. Si no te sale, comprueba que las características de la conexión que has establecido coinciden con las de tu circuito, y que tienes los pines correctamente configurados. A veces sucede que el *hiperterminal* no funciona a altas velocidades de transmisión. En estos casos, el error aparece al configurar el *hiperterminal*, sin siquiera conectar la placa al PC por el puerto

serie. Si esto sucede, prueba a poner una velocidad baja, por ejemplo 9600. Comprueba también que el cable serie está bien conectado. Y si nada de esto funciona, prueba a simular el circuito completo y si aún no funciona, mira el apartado 7.6.4.

Figura 7.33: Conexión al puerto serie

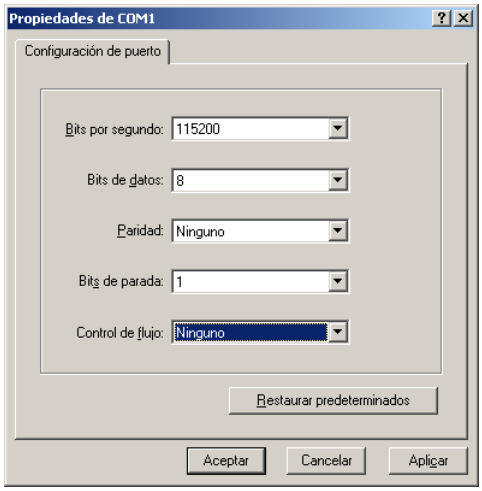


Figura 7.34: Características de la conexión

7.6.4. Cuando la síntesis no funciona *

Puede ser que hayas seguido todos los pasos anteriores y aún así no te funcione el circuito. Las causas de que no funcione pueden ser muy diversas, y por lo tanto es resulta difícil de dar una guía exhaustiva de cómo proceder. Sobre todo es la experiencia la que te ayudará a resolver de manera más rápida estos problemas. Aún así te daremos una serie de puntos que podrás revisar cada vez que no sepas dónde está el error.

Importante: los momentos de búsqueda de fallos pueden ser desesperantes, sobre todo cuando se alargan mucho y los tiempos de entrega se acercan. En estos momentos críticos a veces sucede que estropeamos más que arreglamos. Intentar mantener la calma puede ser un consejo difícilmente alcanzable, pero lo que sí está en tu mano es hacer copias de seguridad. Cada vez que vayas a arreglar algo, **haz una copia de seguridad**. Lo recomendable sería utilizar un sistema de control de versiones, pero como mínimo, deberías de hacer copias de los ficheros que vayas a modificar (incluyendo al nombre del fichero el número de versión, por ejemplo: `nombrefichero_v13.vhd`).

A continuación te damos una serie de puntos que esperamos que te ayuden a encontrar los errores:

7.6.4.1. Comprueba que todo esté bien conectado

Aunque esta recomendación pueda parecer muy evidente, a veces las cosas no funcionan por este tipo de despistes y se pierde mucho tiempo buscándolo en otro sitio. Comprueba que todo esté bien conectado: placa enchufada, interruptor encendido, cables conectados: RS-232, USB,...

7.6.4.2. Comprueba el fichero `.ucf`

Mira el fichero `.ucf` y vuelve a repasar la correspondencia de los pines con los puertos. Comprueba que no estás nombrando los pines de otra placa. Recuerda que en la Nexys2 hay algunos pines que en la placa no están correctamente nombrados, mira las hojas de características de la placa y vuélvelos a revisar.

7.6.4.3. Comprueba el reset del circuito

Haz las siguientes comprobaciones del **reset** de tu circuito:

- Mira de qué pulsador o interruptor viene el reset y verifica que no lo tengas reseteado.
- Comprueba que en el diseño VHDL el reset de los elementos de memoria se active siempre al mismo nivel.
- Si has usado la constante c_on para activar los elementos de memoria (como hemos recomendado), comprueba que el valor de la constante se corresponde con el valor de activación de su pulsador correspondiente. Recuerda que los pulsadores de las placa *XUPV2P* funcionan a nivel bajo y los de la *Nexys2* a nivel alto.

7.6.4.4. Simula el circuito completo

Puede ser que hayas cada uno de los módulos por separado, pero no el circuito completo. Quizá el fallo esté en alguna conexión del módulo de más alto nivel. Simula el circuito completo y revisa bien las formas de onda. Vuelve a repasa las simulaciones de los módulos.

7.6.4.5. Revisa las advertencias del sintetizador

Observa detenidamente los ***warnings*** (advertencias) que da el ISE al sintetizar. Hay algunos que nos pueden ayudar a encontrar el fallo (otros no tienen mucha utilidad). Para ver los *warnings* tienes que pinchar en *View Design Summary* dentro de la subventana *Processes* (punto 1 de figura 7.35) o bien seleccionar la pestaña del resumen del diseño (*Design Summary*) (punto 2 de figura 7.35). En la ventada del resumen del diseño tienes que pinchar en los *warnings* (punto 3 de figura 7.35). También los puedes ver pinchando en la pestaña de *Warnings* de la subventana inferior (punto 4 de figura 7.35), sin embargo, aquí sólo veras los *warnings*, y hay ciertos avisos que pueden ser muy útiles que no se muestran aquí.

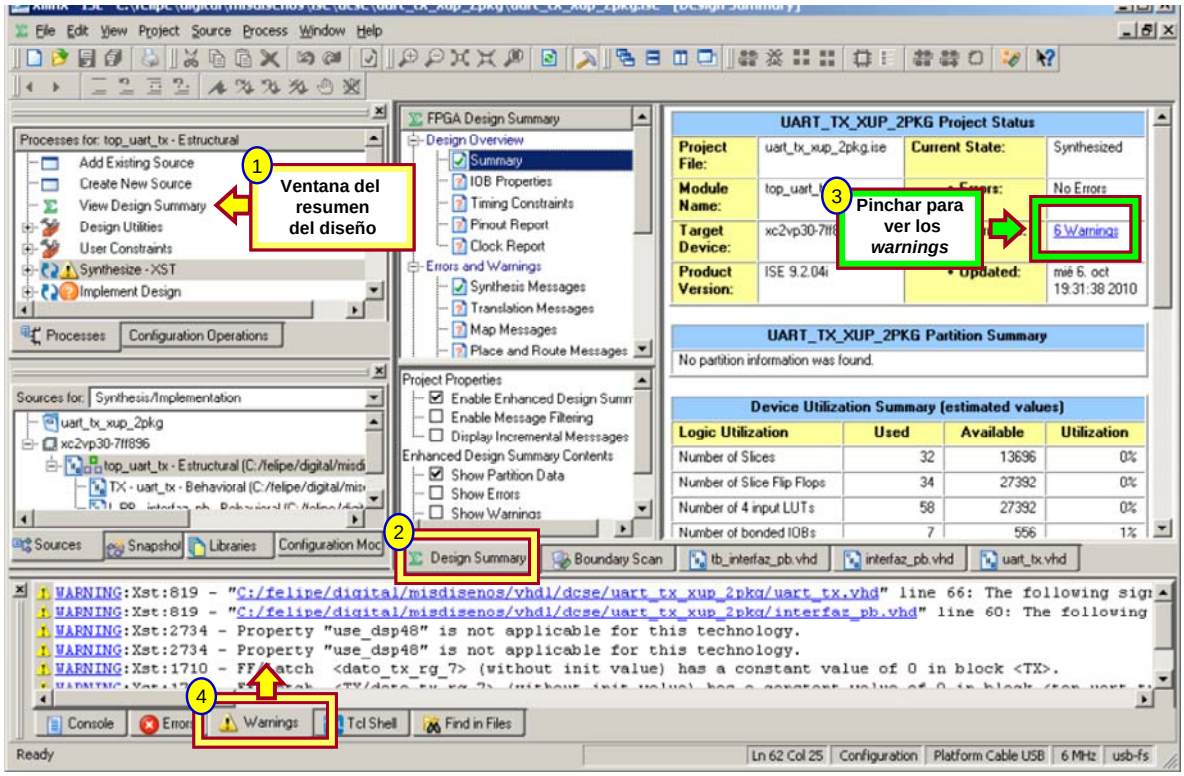


Figura 7.35: Resumen del diseño en el ISE

A continuación se explican el significado de algunos de estas advertencias:

1. Errores en las **listas de sensibilidad**. Las listas de sensibilidad son para simulación (repasa el manual de ED2 [17mach]), pero no se usan en síntesis. Cuando un proceso tiene mal la lista de sensibilidad, puede ocurrir que la simulación no se corresponda con la síntesis. Recuerda que en la lista de sensibilidad se deben incluir las señales que se leen en el proceso[42], aunque si es un proceso con reloj basta con incluir la señal de reloj y las anteriores[43] (el reset habitualmente). Así que comprueba el *warning* que dice:

   Xst:819 - The following signals are missing in the process sensitivty list...

2. Comprueba si hay **registros** que se quedan **con valores constantes**. Esto no siempre implica que sea un error, así que debes de comprobar si se trata de un error o está bien así. Por ejemplo, probablemente al sintetizar el circuito completo te haya salido el siguiente error:

   Xst:1710 - FF/Latch <dato_tx_rg_7> (without init value) has a constant value of 0 in block <TX>.

   Esta advertencia nos avisa que el bit 7 de la señal `dato_tx_rg` que está en el bloque `TX` tiene el valor constante '0'. El nombre del bloque (`TX` en este caso) lo da el nombre que hemos puesto a la referencia o instancia del componente (recuerda la figura 7.31 donde se puso el nombre `TX` al componente `UART_TX`). En nuestro caso tendremos que comprobar por qué el bit 7 de la señal `dato_tx_rg` se queda a cero.

   Que un biestable esté siempre a cero, aparentemente es un error. Para descubrir si realmente se trata de un error tenemos que ir al proceso donde se asigna la señal (fichero `uart_tx.vhd`) y analizar las asignaciones. El proceso es el registro de desplazamiento. En la figura 7.36 se muestra el proceso[44] y en ella podemos apreciar que el bit 7 de `dato_tx_rg` se asigna sólo en las dos primeras sentencias, mientras que en la última sentencia se asignan todos los bits menos el 7. Esto es porque es un registro de desplazamiento a la derecha y por tanto `dato_tx_rg(7)` mantiene su valor. En la primera sentencia (en el reset) se asigna un cero y en la segunda se asigna `dato_tx_in(7)`. Por lo tanto habrá que seguir investigando qué valores tiene `dato_tx_in(7)`.

```
P_CargaDesplaza: Process(rst, clk)
begin
  if rst = c_on then
    dato_tx_rg <= (others => '0');          ---> dato_tx_rg(7) <= '0'
  elsif clk'event and clk = '1' then
    if cargadato = '1' then
      dato_tx_rg <= dato_tx_in;             ---> dato_tx_rg(7) <= dato_tx_in(7)
    elsif desplaza = '1' then
      dato_tx_rg (6 downto 0) <= dato_tx_rg (7 downto 1);  ---> dato_tx_rg(7)
    end if;                                                    no recibe valor,
  end if;                                                      guarda el que tenía
end process;
```

Figura 7.36: Asignación de la señal `dato_tx_rg`

[42] Revisa el capítulo 3 del manual de ED2 [17mach]

[43] Revisa el capítulo 5 del manual de ED2 [17mach]

[44] Probablemente no hayas realizado el proceso igual, pero fíjate en estos ejemplos para entender lo que se intenta explicar, y luego compáralo con tu diseño.

El puerto `dato_tx_in` está conectado con el puerto `caracter` del `INTERFAZ_PB` (figura 7.30). Así que tendremos que analizar qué valores recibe el puerto `caracter`. En la figura 7.37 se muestra el proceso donde se asigna[45].

```
P_envio_orden: Process(pulso_0, pulso_1, pulso_2, pulso_3, tx_ocupado)
begin
  if tx_ocupado = '1' then
    transmite <= '0';
    caracter <= (others => '0');  ------------------------> caracter(7) <= '0'
  else
    transmite <= '0';
    caracter <= (others => '0');  ------------------------> caracter(7) <= '0'
    if pulso_0 = '1' then
      transmite <= '1';
      caracter  <= x"61";  -- "01100001", letra 'a' ------> caracter(7) <= '0'
    elsif pulso_1 = '1' then
      transmite <= '1';
      caracter  <= x"6C";  -- "01101100", letra 'l' ------> caracter(7) <= '0'
    elsif pulso_2 = '1' then
      transmite <= '1';
      caracter  <= x"6F";  -- "01101111", letra 'o' ------> caracter(7) <= '0'
    elsif pulso_3 = '1' then
      transmite <= '1';
      caracter  <= x"68";  -- "01101000", letra 'h' ------> caracter(7) <= '0'
    end if;
  end if;
end process;
```

Figura 7.37: Asignación del puerto de salida `caracter`

Observando el proceso nos daremos cuenta que el bit 7 de `caracter` siempre recibe un cero. Esto es una casualidad, debido a que las cuatro letras que asignamos tienen el bit más significativo a cero. El sintetizador se ha dado cuenta de esto, y en vez de usar un biestable para este bit, lo ha reducido a una constante y lo avisa con ese *warning*. En otro caso podría ser debido por un error nuestro al codificar, pero en este caso, debido a la particularidad de las asignaciones, la simplificación es correcta.

3. **Señales que no se usan**. Normalmente declarar una señal sin usarla puede significar que te has olvidado de usarla o que inicialmente pensabas que la ibas a usar y te has dado cuenta que no la necesitabas. En el primer caso, este *warning* te está recordando que te falta por hacer algo en el circuito y puede ser que sea por esto que no funcione. En el segundo caso deberías de borrar o como mínimo comentar la declaración. Es importante tener el código ordenado y quitar cosas inútiles que dificulten la comprensión del código. El aviso que da el ISE para estas señales que se declaran pero no se usan es el siguiente (en este ejemplo la señal se llama `s_inutil`):

   Xst:1780 - Signal <s_inutil> is never used or assigned.

4. **Señales que se asignan pero que no se usan**. Cuando has asignado valor a una señal pero no usas esa señal, seguramente sea que te has olvidado darle uso. En la figura 7.38 se representa este caso. Como puedes ver no tiene sentido crear una señal y asignarle valor para nada.

[45] De la misma manera que el caso anterior, tu proceso puede ser diferente

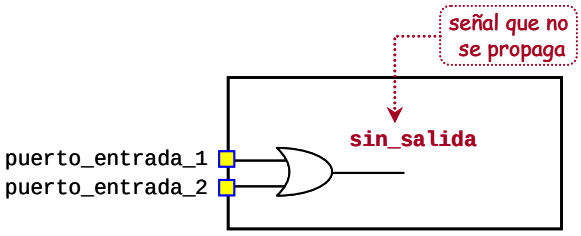


Figura 7.38: Esquema de un circuito con una señal que no da valor

El *warning* que da el sintetizador ISE es el siguiente:

Xst:646 - Signal <sin_salida> is assigned but never used.

5. **Señales que se usan sin recibir valor**. Es el caso contrario al anterior, es bastante grave que una señal que estés usando no le hayas asignado valor. Sería como poner un cable al aire y seguramente implique que te hayas olvidado de incluir la sentencia que le da valor. La figura 7.39 representa este caso, donde a la señal `sin_asignar` no se le ha asignado nada.

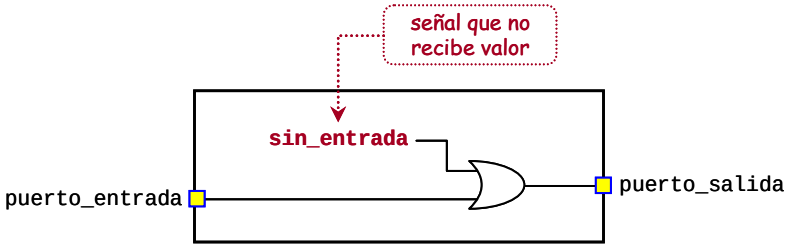


Figura 7.39: Esquema de un circuito con una señal que no ha recibido valor

En este caso el *warning* es el siguiente, y el sintetizador la pone a valor cero.

Signal <sin_entrada> is used but never assigned. Tied to value 0.

6. **Creación de *Latches***. Cuando el sintetizador avisa de que ha creado un *latch*[46] para una señal conviene que comprobar que realmente queríamos generar el *latch*. Aunque la creación de un *latch* no necesariamente implica que sea un error, en general se recomienda diseñar circuitos síncronos con el reloj, haciendo que los elementos de memoria sean activos por el flanco del reloj. Esto es una recomendación de diseño porque son más seguros y dan menos problemas, pero no implica que sea erróneo.

   Además de esto, muchas veces se crean *latches* involuntariamente. Por ejemplo, cuando una señal combinacional no la asignamos en todas las alternativas, la señal tiene que guardar su valor y para ello genera un *latch*. Implementar la señal con un *latch* en vez de con un circuito combinacional puede producir un comportamiento indeseado en el circuito. Así que cuando veas este *warning* comprueba que la señal se asigna en todas las alternativas, prestando atención en que las sentencias `if` terminen en `else` (y no en `elsif`). Revisa los capítulos 4 y 5 del manual de ED2 [17mach] si no entiendes esto último.

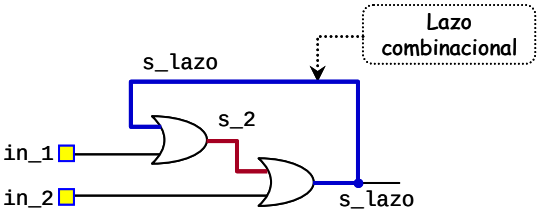
   El aviso que da el ISE se muestra a continuación, en el que indica que se ha generado un *latch* para la señal `s_latch`.

   Xst:737 - Found 1-bit latch for signal <s_latch>.

7. Formación de **lazos combinacionales**. Cuando aparece esta advertencia hay que descubrir qué la ocasiona y solucionarlo. En un circuito síncrono con FPGA no debería de haber

[46] Los *latchs* se explican en el apartado 2.5 y más en profundidad el capítulos 4 y 5 del manual de ED2 [17mach]

bucles combinacionales. Un bucle combinacional se forma cuando el recorrido de una señal se propaga hacía sí misma sin que hayan elementos de memoria en medio. En la figura 7.40 se muestra el esquema de un lazo combinacional y el VHDL que lo genera.

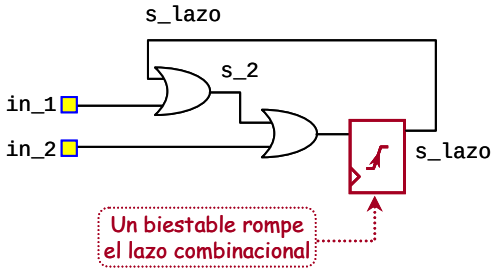



```
s_2    <= in_1 or s_lazo;
s_lazo <= in_2 or s_2;
```

Figura 7.40: Esquema de un circuito con lazo combinacional y el código VHDL que lo forma

Fíjate que las señales `s_2` y `s_lazo` forman el lazo combinacional. Estos lazos provocarían un funcionamiento asíncrono y errático por los retardos de las puertas y conexiones. Por lo tanto hay que evitarlos, para ello hay que entender la funcionalidad del circuito e intentar buscar una descripción alternativa. La manera habitual de evitarlos es poner un biestable en medio, aunque hay que hacerlo con cuidado, siendo necesario comprobar que el nuevo circuito cumple la funcionalidad deseada.

La figura 7.41 muestra cómo se puede romper el lazo combinacional de la figura 7.40. Observa que este lazo se podría romper también por la señal `s_2`, en vez de por la señal `s_lazo`. La funcionalidad de estos circuitos sería diferente, por lo que se debe de comprobar si el circuito se comporta como queremos.




```
s_2 <= in_1 or s_lazo;

P: Process (rst,clk)
begin
  if rst = c_on then
    s_lazo <= '0';
  elsif clk'event and clk='1' then
    s_lazo <= in_2 or s_2;
  end if;
end process
```

Figura 7.41: Esquema que muestra cómo se puede romper el lazo combinacional de la figura 7.40

El aviso del sintetizador del ISE se muestra a continuación. Sin embargo, si miras el aviso por la ventana del resumen del diseño (punto 2 de la figura 7.35), no se muestra la señal que forma el lazo. Tienes que mirarlo en la pestaña de *Warnings* de la subventana inferior (punto 4 de figura 7.35). Aún así, la información no siempre es muy clara y hay que investigar un poco para encontrar el lazo.

Xst:2170 - Unit interfaz_pb : the following signal(s) form a combinatorial loop: s_lazo.

Por último indicar que los lazos combinacionales son más difíciles de detectar cuando se forman entre varios módulos de un diseño estructural. Veámoslo con un ejemplo similar al del transmisor de la UART pero más sencillo. Tenemos un circuito estructural con dos módulos (figura 7.42).

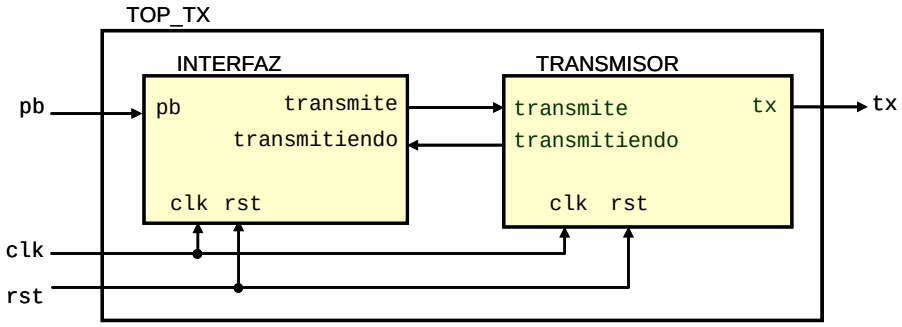


Figura 7.42: Esquema de ejemplo de circuito estructural para mostrar lazos combinacionales entre módulos

Un módulo es un transmisor que cuando recibe la orden de transmitir (`transmite='1'`) comienza la transmisión por el puerto `tx`. Durante la transmisión pone la señal `transmitiendo` a valor 1. Este circuito tiene dos estados: `e_init` y `e_tx`. En el primero está en reposo esperando la orden, y en el segundo está transmitiendo, al terminar de transmitir, vuelve al estado de reposo.

El otro módulo es un interfaz con un pulsador `pb`. Este circuito detecta el flanco del pulsador generando la señal `pulso_pb`. Cuando hay flanco (`pulso_pb='1'`), ordena el inicio de la transmisión al transmisor. En caso de que el transmisor esté transmitiendo (`transmitiendo='1'`) el interfaz no ordenará la transmisión.

Podríamos describir los procesos que asignan las salidas `transmite` y `transmitiendo` como se muestran en la figura 7.43.

INTERFAZ

```
...
P_I: Process (transmitiendo, pulso_pb)
begin
  if transmitiendo = '1' then
    transmite <= '0';
  elsif pulso_pb='1' then
    transmite <= '1';
  else
    transmite <= '0';
  end if;
end process;
...
```

TRANSMISOR

```
...
P_T: Process (estado, transmite)
begin
  case estado is
    when e_init =>
      if transmite = '1' then
        transmitiendo <= '1';
      else
        transmitiendo <= '0';
      end if;
    when e_tx =>
      transmitiendo <= '1';
  end case;
end process;
...
```

transmite se pone a **'0'** cuando **transmitiendo='1'**

transmitiendo se pone a **'1'** cuando **transmite='1'**

transmite = '1' ⇨ **transmitiendo ='1'** ⇨ **transmite = '0'**

Por la rapidez, el estado no habrá cambiado, **estado=e_init** y **pulso_pb** todavía estará a **'1'**

transmite = '0' ⇨ **transmitiendo ='0'** ⇨ **transmite = '1'**

Figura 7.43: Procesos en módulos distintos que forman un lazo combinacional con las señales `transmite` y `tranmitiendo`.

Si observas los procesos, ambos son combinacionales. La señal `transmitiendo` depende de la señal `transmite` y viceversa. Por lo tanto ambas forman un lazo combinacional. Este lazo combinacional haría que en cuanto la señal `transmite` se pusiese a uno se volvería a poner a cero casi instantáneamente debido a que `transmitiendo` se pondría

a uno. Tardaría el tiempo de retardo de las señales y las puertas. Sin embargo, las señales no serían estables y volverían otra vez a los valores iniciales porque `pulso_pb` seguiría a uno y el estado no habría cambiado, ya que por haber sido tan rápido el cambio de las señales, aún no habría llegado el flanco de reloj que haría cambiar a los biestables. Además esto podría hacer que el circuito fuese inestable porque no cumplirse los tiempos de *setup* y *hold* de los biestables, provocando metaestabilidad en los biestables (esto se explicó en ED2).

¿Cómo podríamos solucionar este problema y romper el lazo combinacional? una primera solución podría ser registrar[47] las salidas `transmite` y/o `transmitiendo`. La otra solución es hacer que la señal `transmitiendo` dependa sólo del estado y no de la entrada `transmite`. Si recuerdas de ED2, esto es hacer que la máquina de estados sea una máquina de Moore en vez de Mealy (el proceso de la figura 7.43). La figura 7.44 muestra el proceso como máquina de Moore, con esto se rompe el lazo combinacional.

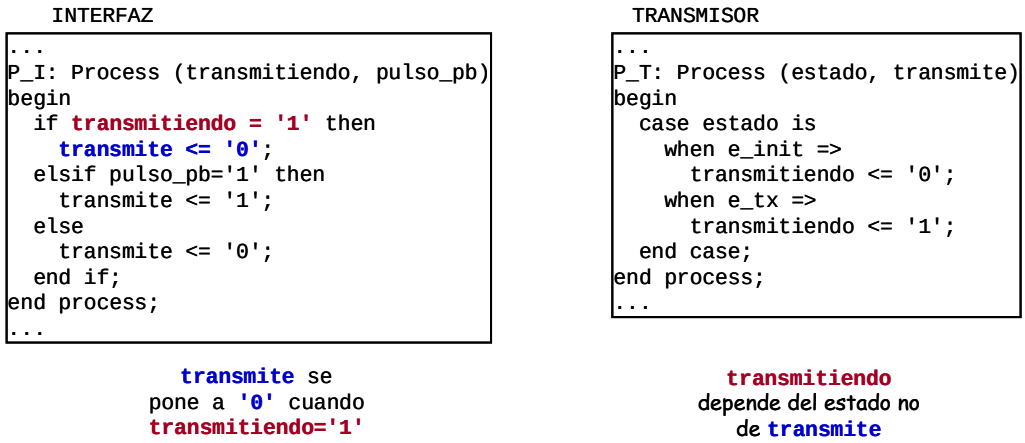


Figura 7.44: El proceso del transmisor como máquina de Moore para romper el lazo combinacional.

8. **Simplificación de lógica**. El sintetizador simplifica la lógica que no es útil en el circuito final. Tenemos que estar muy atentos a estas simplificaciones porque implica que lo que hemos querido codificar no se va implementar como esperamos. Por ejemplo, supón que en al sintetizar el transmisor de la UART tenemos un aviso que indica lo siguiente[48]:

   Xst:2679 - Register <baud> in unit <uart_tx> has a constant value of 0 during circuit operation. The register is replaced by logic.

   De la figura 7.12 y recordando lo que hace nuestro transmisor podemos deducir que si la señal `baud` tiene un valor constante cero, nuestro transmisor no funcionará, pues es el que hace cambiar de estados y hacer los desplazamientos del registro de desplazamiento. Efectivamente, el que la señal `baud` valga cero el la causante de que por ejemplo la señal `desplaza` tenga el mismo aviso que `baud`. Y también la responsable de los avisos:

   Xst:1799 - State e_bits_dato is never reached in FSM <estado_tx>.

   Xst:1799 - State e_bit_fin is never reached in FSM <estado_tx>.

   Es decir, nos está diciendo que nunca se llega a los estados `e_bits_dato` y `e_bit_fin`. Lo que es lógico si `baud` siempre se queda a cero.

[47] *Registrar una señal* es lo mismo que poner un biestable en esa señal.

[48] Curiosamente este aviso no está catalogado cono *warning* sino como *info*, por lo que no se ve en la suventana de *Warnings* (la que está abajo, el punto 4 de figura 7.35).

Otro *warning* nos dice que:

Xst:646 - Signal <dato_tx_rg<7:1>> is assigned but never used.

Esta advertencia también es lógica, pues `dato_tx_rg` es el registro de desplazamiento., que como la señal `desplaza` se queda a cero, los bits del 7 al 1 nunca van a ser usados.

Por tanto, las advertencias del sintetizador pueden darnos información muy valiosa. Estando atentos a ésta podemos detectar en qué falla el circuito. En este caso, era un fallo del divisor de frecuencia y este fallo también lo hubiésemos detectado mediante simulación, pero cuando no encuentras el fallo, es importante buscar cualquier fuente de información.

9. Otros avisos. No podemos dar una lista exhaustiva de todos los *warnings* e *infos* que da el sintetizador, además de que haría que este manual fuese muy aburrido. Con esta introducción ya tienes suficiente información para investigar por ti mismo el significado de otros avisos. Hay avisos que no son importantes y los verás en casi todos los diseños, como:

   Xst:2734 - Property "use_dsp48" is not applicable for this technology.

   Así que con atención y la experiencia que vayas adquiriendo irás aprendiendo a identificar el significado de los avisos y su importancia.

7.6.4.6. *Muestra información del circuito por los leds*

Cuando el circuito no hace nada que podamos ver, como por ejemplo el circuito del transmisor de la UART, no tenemos información de si el circuito está haciendo algo. En estos casos los leds pueden ser una fuente de información muy valiosa. Puedes sacar señales que te ayuden a conocer el estado interno. Puedes también codificar los valores de los estados en los leds, para ver si se queda atascado en algún estado. Ten cuidado de no sacar señales cuya activación sea muy corta, porque si es así, no podrás apreciar la activación. Al sacar señales por los leds ten en cuenta las características de los leds de la placa, esto es, si funcionan a nivel alto o bajo. Mejor es que uses las constantes `c_on` y `c_off`.

7.6.4.7. Crea un nuevo proyecto en el ISE

A veces el proyecto del ISE se corrompe y no actualiza los cambios que hagas en el circuito. En estos casos, el fichero `.bit` generado no cambia y corresponde a una versión anterior. La solución es crear un nuevo proyecto e importar los ficheros `.vhd` del anterior. Puede ser desesperante que después de horas de intentarlo todo, sea esto lo que haya originado el fallo (aunque por otra parte sea un alivio haber encontrado el fallo).

7.7. Diseño del receptor de la UART

En este apartado se darán las indicaciones para diseñar el receptor de la UART. En los apartados anteriores se ha dado mucha información con el objetivo de que puedas ser más independiente en los diseños que se propongan a partir de ahora. Así que no dudes en repasar los conceptos anteriores en caso de que te encuentres con dificultades.

Con todo lo que hemos visto sobre el funcionamiento del protocolo RS-232, después de haber diseñado el transistor e incluso haber diseñado un receptor para simulación (apartado 7.5.4), no debería ser necesaria mucha explicación para que puedas diseñar un receptor sintetizable.

La entidad que queremos diseñar tiene el aspecto mostrado en la figura 7.45:

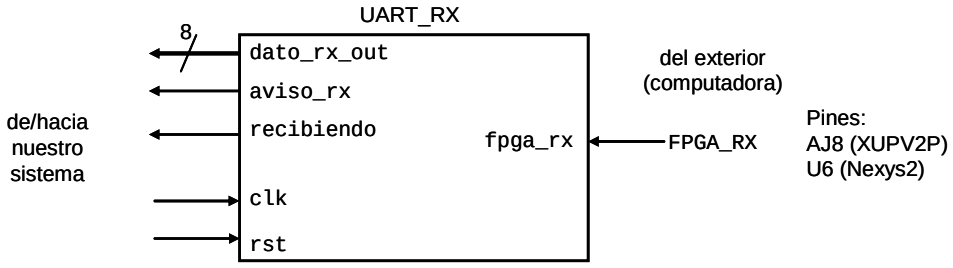


Figura 7.45: Entradas y salidas del receptor

Las constantes del diseño serán las mismas que usamos para el transistor (tabla 7.2) y de hecho usaremos los mismos paquetes.

Las especificaciones de los puertos son las siguientes:

Puerto	bits	I/O	Descripción
`rst`	1	I	Señal de reset asíncrono, su nivel dependerá de `c_on`
`clk`	1	I	Señal de reloj de la placa. La frecuencia del reloj se indica en la constante `c_freq_clk`.
`fpga_rx`	1	I	Trama que se recibe en serie desde la computadora, sigue el formato RS232
`aviso_rx`	1	O	Aviso de que se ha recibido un nuevo dato y que está disponible en `dato_rx_out`. El aviso se dará poniendo la señal a '1' durante un único ciclo de reloj.
`dato_rx_out`	8	O	Proporciona los 8 bits del dato que se ha recibido. Este dato es sólo será válido desde que `aviso_rx` valga '1' y mientras `recibiendo` sea '0'
`recibiendo`	1	O	Cuando vale '1' indica que el módulo se encuentra recibiendo una trama y por tanto el valor del dato `dato_rx_out` no es válido

Tabla 7.6: Puertos del receptor de la UART

En la figura 7.46 se muestra cómo debería ser el cronograma de las salidas del receptor al terminar una recepción (fíjate dónde está el bit de fin), posteriormente con la línea en reposo, y por último al comienzo de una nueva recepción (fíjate dónde está el bit de inicio). A partir de la figura se observa que el dato recibido (`dato_rx_out`) sólo será válido a partir del aviso de la señal `aviso_rx` y mientras que la señal `recibiendo` se mantenga a '0'. Cuando la señal `recibiendo` se ponga a '1' el dato no será válido. Nunca podrán estar simultáneamente a '1' las señales `recibiendo` y `aviso_rx`.

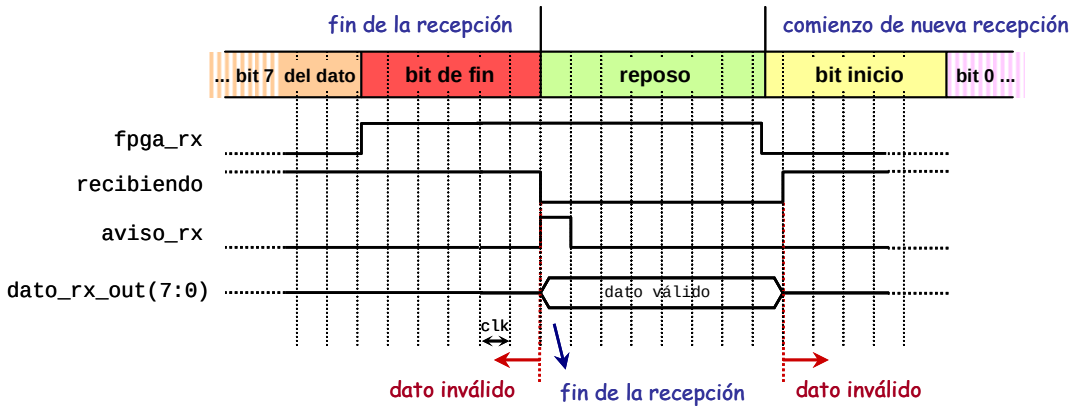


Figura 7.46: Cronograma de salidas del receptor al terminar la recepción y el comienzo de una nueva

A partir de aquí, intenta hacer el receptor por ti mismo. Si ves que tienes dudas, o para comparar tu solución con una propuesta, consulta las guías que se dan a continuación.

El diseño del receptor se puede realizar de manera parecida al del transmisor. Una primera aproximación al diagrama de bloques del diseño se muestra en la figura 7.47.

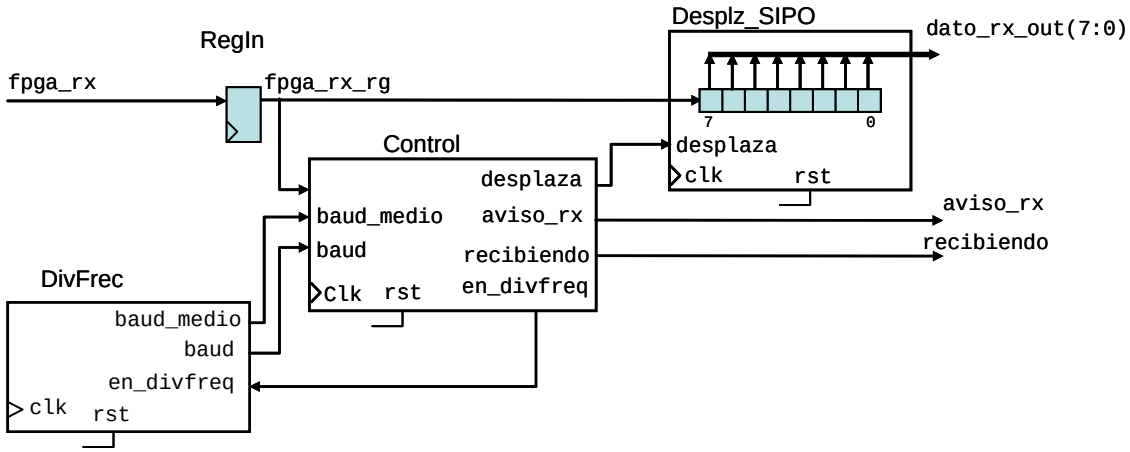


Figura 7.47: Diagrama de bloques preliminar del receptor

A continuación se describen brevemente cada uno de los bloques.

7.7.1. Registro de desplazamiento

El registro de desplazamiento (`Desplz_SIPO`) es un registro al que se le van cargando los datos en serie y los devuelve en paralelo (*serial-in parallel-out*: SIPO). Éste es el registro opuesto al que teníamos en el transmisor, que cargaba en paralelo y devolvía en serie (PISO). Como el primer bit que se recibe es el bit 0, si la carga serie se hace por el bit más significativo del registro y se hacen desplazar los bits hacia la derecha [bit n→ bit (n-1)], en el último desplazamiento, el bit 0 recibido estará en el bit 0 del registro, y estarán todos los bits ordenados. La carga y el desplazamiento se realizan bajo la orden de la señal `desplaza` originada en el bloque `control`

7.7.2. Registro de entrada

La entrada de la comunicación serie `fpga_rx` se registra en `fpga_rx_rg` (módulo `RegIn` de la figura 7.47). Como `fpga_rx` es una señal asíncrona es conveniente registrarla, y para evitar meta-estabilidad se puede incluso registrar dos veces poniendo dos registros en cascada (recuerda la figura 2.11).

7.7.3. Divisor de frecuencia

Para el receptor, el divisor de frecuencia debe sincronizarse con la trama que se recibe. Esto es así porque el receptor no dirige la transmisión, como sucedía con el transmisor. El divisor de frecuencia se mantiene inactivo con la cuenta a cero hasta que la entrada `en_divfreq` se pone a '1'. El divisor de frecuencia saca dos señales: `baud` y `baud_medio`. `baud` marca la transición del estado, y `baud_medio` marca el punto medio de cada bit de la transmisión. En este punto medio es donde se evalúa el valor del bit recibido, evitando evaluar el bit cerca de las transiciones donde se puede tomar el valor del bit contiguo. Esto es similar a lo que hicimos con el receptor para simulación, recuerda la figura 7.24.

En la figura 7.48 se muestra el cronograma del divisor de frecuencia con una cuenta de ejemplo de 20 (en realidad la cuenta es mucho más larga, y viene indicado por la constante `c_cont_baud` calculada en el código 7.2).

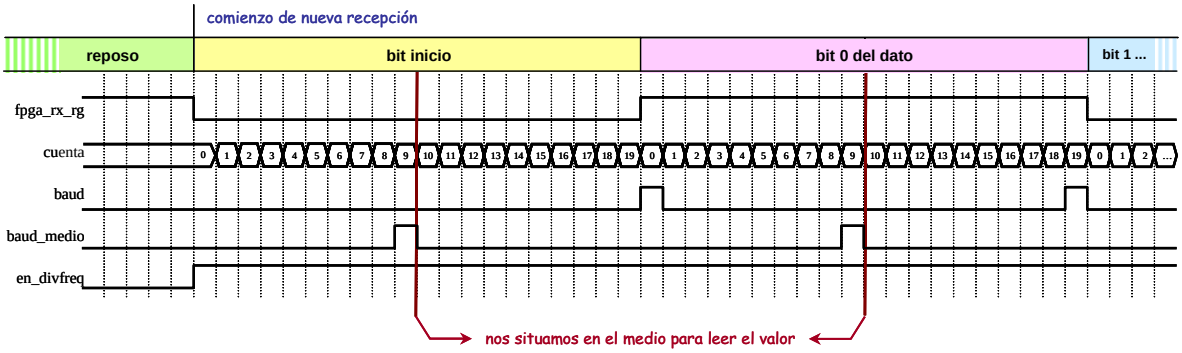


Figura 7.48: Cronograma del divisor de frecuencia

7.7.4. Bloque de control

El bloque de control es una máquina de estados similar a la del transmisor (apartado 7.4.2). El cambio de estado también viene provocado por `baud`. La orden de desplazamiento (`desplaza`) se dará cuando se está en el estado de recepción de dato (`e_bits_dato`) y llega la señal `baud_medio`.

7.8. Banco de pruebas del receptor de la UART

Para hacer un banco de pruebas tenemos que crear un modelo de simulación de las entradas. En el caso del receptor, sólo tenemos tres entradas (figura 7.45), siendo dos de ellas el reloj y el reset. La otra entrada es el puerto `fpga_rx`, que se corresponde con el dato serie que se recibe con el protocolo RS-232. Por tanto, para realizar el banco de pruebas tendríamos que realizar un proceso de simulación que modelase un envío RS-232. Habría que contar los tiempos de cada bit, y enviar el bit de inicio, los 8 bits del dato, el bit de fin y finalmente poner la línea en reposo. Con todo lo que hemos visto sobre el protocolo RS-232 y sobre simulación, no deberías de tener muchas dificultades para realizar un banco de pruebas de este tipo. La figura 7.49 muestra un esquema de cómo podríamos hacer el banco de pruebas del receptor. Observa que es similar a banco de pruebas del transmisor 7.22.

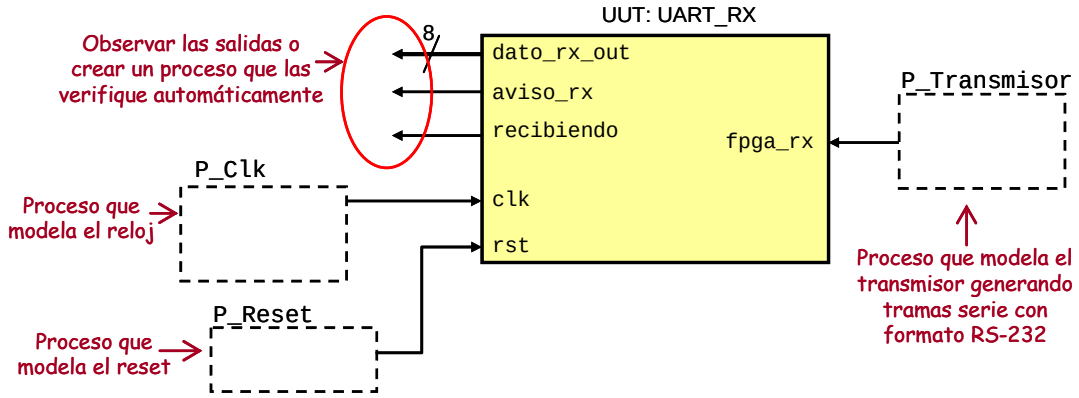


Figura 7.49: Esquema del banco de pruebas del receptor, que incluye un proceso que modela el transmisor descrito a alto nivel

Otra opción para el banco de pruebas es utilizar el transmisor que ya hemos realizado y su banco de pruebas para generar la trama RS-232. Con esto nos ahorraríamos realizar el modelo de alto nivel del transmisor que genera la trama RS-232. Como ya hemos comprobado que el transmisor funciona, en este banco de pruebas sólo estaríamos probando el receptor, que es lo que queremos probar. La figura 7.50 muestra el esquema de este banco de pruebas. La ventaja de hacer este banco de pruebas es que tardamos menos en diseñarlo, la desventaja es que la simulación será más lenta porque el modelo del transmisor es más complejo que si fuese un modelo para simulación.

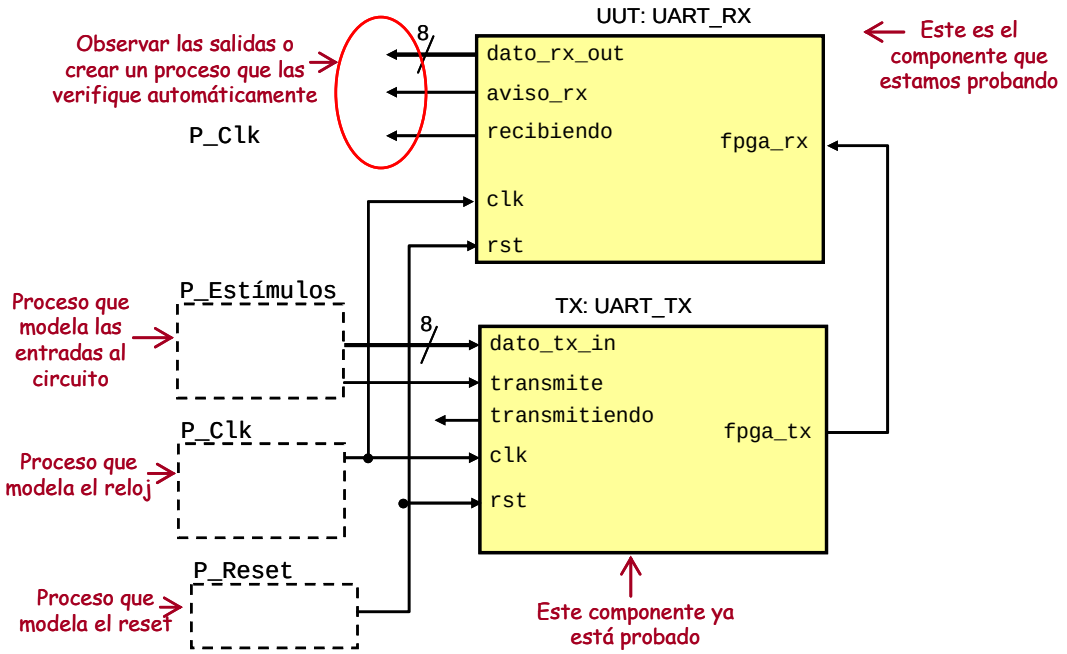


Figura 7.50: Esquema del banco de pruebas del receptor, que incluye el transmisor sintetizable

Ahora, elige uno de los dos bancos de pruebas y comprueba que las salidas `dato_rx_out`, `aviso_rx` y `recibiendo` son correctas. Probablemente no te salga bien a la primera y tendrás que analizar las señales y estados internos del receptor. Realiza esto por ti mismo para afianzar el análisis de la simulación explicado en el apartado 7.5. Consulta este apartado si no sabes cómo proceder a la comprobación de las formas de ondas.

7.9. Implementación de la UART completa

Si has probado el receptor, ahora lo podrás unir con el transistor e implementarlos juntos en un sólo diseño. Vamos a realizar un diseño que transmita lo recibido por el receptor, consiguiéndose un eco en el *hiperterminal*. Para ello, cada vez que se reciba un dato por el receptor tendríamos que dar la orden de transmitir. Podemos utilizar el puerto `aviso_rx` del receptor para dar la orden al transmisor por el puerto `transmite`, puesto que ambas funcionan con un ciclo de reloj. Además, cuando `aviso_rx` está activo, el dato está disponible en `dato_rx_out`, por lo que no habría que hacer nada más que conectarlos como muestra la figura 7.51.

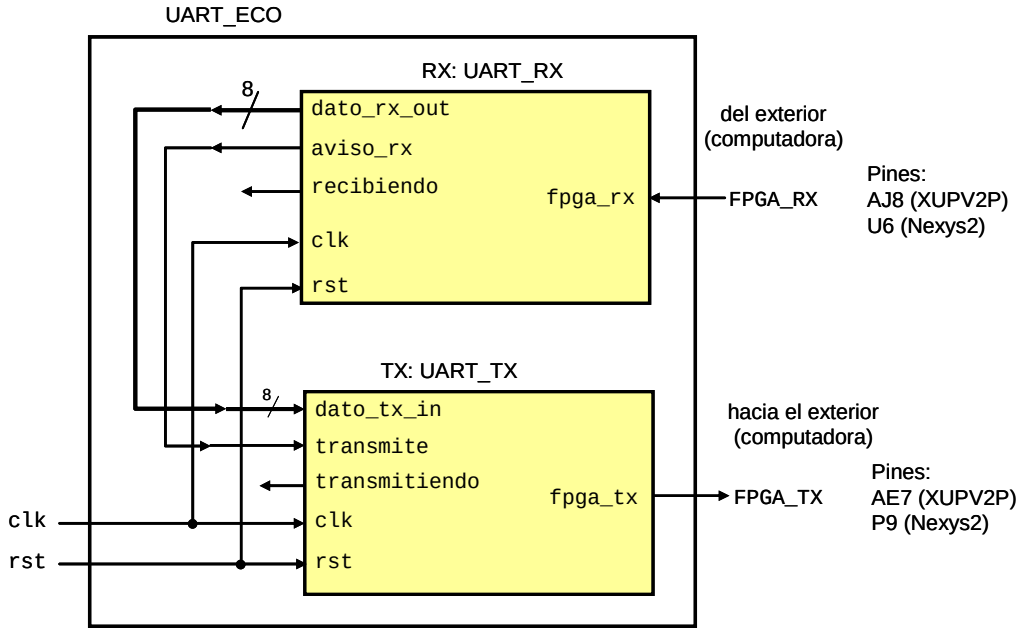


Figura 7.51: Circuito que transmite lo que recibe con protocolo RS-232

Implementas el diseño y observa si tienes eco. Ten en cuenta que los *hiperterminales* normalmente no tienen eco, esto es, no muestran lo que escribes (a no ser que lo configures), por tanto si estás viendo lo que escribes, probablemente lo tengas bien. Prueba a escribir sin el cable conectado o con la FPGA desprogramada a ver si ves eco.

Si obtienes caracteres distintos a los que has enviado, revisa la configuración del *hiperterminal*. Si no te sale bien, comprueba los pasos del apartado 7.6.4 o realiza un banco de pruebas del circuito completo.

8. Circuitos aritméticos

En esta práctica se introducirá el diseño de módulos aritméticos en VHDL. La implementación de este tipo de módulos es de mucha utilidad, ya que una de las aplicaciones de los dispositivos programables como las FPGA es acelerar el cómputo. Sin embargo, para la implementación de la mayoría de las operaciones matemáticas necesitamos describir cómo es el hardware. Es decir, no podremos utilizar el operador de división entre dos señales enteras, sino que tendremos describir cómo es el hardware que implementa la división. Otras operaciones más sencillas como la suma, la resta y, a veces la multiplicación entera, habitualmente las implementa el sintetizador. Así que aunque para estas operaciones normalmente no tenemos que describir cómo se implementa en hardware, es importante saber entender cómo funcionan para evitar errores de diseño[49].

Debemos ser cuidadosos al implementar las operaciones matemáticas en VHDL. Por un lado se deben elegir los **tipos numéricos** correctamente y establecer **operaciones consistentes** entre las señales y variables. Por otro lado, como ya se ha dicho, se debe tener en cuenta que operaciones complejas como la división no son inmediatas en hardware y normalmente se debe especificar qué algoritmo la realizará.

Es importante que no olvidemos que las señales de un circuito (hardware) no son más que un grupo de cables eléctricos. Haciendo una abstracción hemos agrupado y ordenado estos cables, considerando que forman una señal con un número determinado de bits. Y que incluso hemos determinado que esa señal es de un tipo numérico determinado. Así que aunque digamos que una señal es de tipo entero con signo, tenemos que recordar que esa señal no es más que un grupo de cables al que le damos un significado.

El tipo de datos que habitualmente hemos usado para vectores es `std_logic_vector`. Para este tipo de datos el VHDL no tiene implementados operadores aritméticos[50]. Por ejemplo, si queremos realizar una suma de señales `std_logic_vector` debemos implementar nosotros el sumador. Esto tiene sentido, ya que con el tipo `std_logic_vector` no se especifica qué número representa, es decir no sabemos si se corresponde con un binario puro, o si tiene signo, y en ese caso, si se la representación es en complemento a dos, signo magnitud, ... Sin embargo, si usamos vectores de tipo `unsigned` ó `signed` ya estamos especificando la representación y el sintetizador podrá implementar algunas operaciones por nosotros.

En este capítulo veremos los paquetes matemáticos necesarios para implementar las operaciones. Posteriormente veremos cómo trabajar con los tipos `unsigned` y `signed` para realizar operaciones de suma y resta. A continuación verémos cómo describir una multiplicación entera y analizaremos con cierto detalle las ventajas e inconvenientes de algunas implementaciones. Por último se introducirá cómo se podría implementar la operación de la división entera.

---

[49] En la referencia [19mach] se explica cómo describir el hardware de sumadores y restadores usando esquemáticos.

[50] Esto depende del paquete. El paquete `numeric_std` no las permite, pero el paquete `std_logic_unsigned` sí las permite, aunque esto puede llevar a error. Por eso, como veremos, se recomienda el paquete `numeric_std`.

8.1. Paquetes matemáticos

Para usar los tipos `unsigned` y `signed` debemos usar ciertos paquetes (*packages*) que definen estos tipos y las operaciones. Existen dos paquetes que implementan estos tipos el `IEEE.NUMERIC_STD` y el `IEEE.STD_LOGIC_ARITH`. El primero es estándar y su uso está más recomendado[51], aunque ambos se pueden utilizar. Así pues, cambiaremos la cabecera de nuestro fichero a como se muestra en el código 8.1:

```
library IEEE;
  use IEEE.STD_LOGIC_1164.ALL;
  use IEEE.NUMERIC_STD.ALL;               -- usaremos este
```

*Código 8.1: Llamada al paquete `numeric_std`, **recomendado***

Y quitaremos las que Xilinx pone por defecto:

```
library IEEE;
  use IEEE.STD_LOGIC_1164.ALL;
  use IEEE.STD_LOGIC_ARITH.ALL;       -- esta no usaremos
  use IEEE.STD_LOGIC_UNSIGNED.ALL;    -- esta no usaremos
```

*Código 8.2: Llamada a los paquetes `std_logic_arith` y `std_logic_unsigned`, **no recomendado**, usaremos el del código 8.1*

De todas maneras es correcto usar ambos paquetes, pero sí es importante que todos los módulos de tu diseño usen los mismos paquetes y **no mezclar** `numeric_std` con `std_logic_arith`.

Si quieres echar un vistazo a estos paquetes, suelen estar en el directorio de instalación de Xilinx, probablemente lo encontrarás en *C:\Xilinx92i\vhdl\src\ieee*. Allí están todos los ficheros `.vhd` de los paquetes. Dentro de ellos puedes mirar las declaraciones de las funciones, tipos, constantes, ... El VHDL de estos paquetes puede ser un poco complicado. Ten cuidado de no modificar estos ficheros, si los vas a ver a menudo, cópialos en otro directorio. De todos modos estos ficheros son fáciles de encontrar en internet.

8.2. *Tipos* unsigned *y* signed

Ya hemos trabajado con los tipos `unsigned` y `signed` (apartado 4.2 y capítulo 6 de la referencia [17mach]). A diferencia de los `integer` que son datos escalares, los `signed` y `unsigned` son vectores, cuyos elementos representan los bits de la señal.

El uso de estos tipos se recomienda frente al uso de enteros. Aunque parezca más fácil utilizar los tipos enteros, el uso de éstos puede hacer que cometamos errores. Por otro lado, el uso de tipos vectoriales nos facilita la descripción de muchas operaciones habituales en los circuitos digitales, como por ejemplo el desplazamiento, el uso de máscaras o la asignación de bits concretos. Además, nos da una visión más próxima al hardware que estamos describiendo, ya que cada elemento del vector se corresponde con un cable del circuito.

Una señal de tipo `unsigned` representa un número entero sin signo, por tanto se corresponde con un número entero en binario puro. Mientras que una señal de tipo `signed` representa un número entero con signo representado en complemento a dos.

[51] El uso del paquete `std_logic_arith` puede dar lugar a ambigüedades y errores. Si quieres profundizar consulta el siguiente enlace:
http://www.synthworks.com/papers/vhdl_math_tricks_mapld_2003.pdf

En los siguientes ejemplos, para facilitar la identificación de los tipos de señales usaremos las siguientes terminaciones en los nombres de las señales:

- `_us`: señales de tipo `unsigned`
- `_s`: señales de tipo `signed`
- `_slv`: señales de tipo `std_logic_vector`
- `_sl`: señales de tipo `std_logic`
- `_int_ps`: señales de tipo `integer` con rango positivo (sin signo)
- `_int`: señales de tipo `integer` con rango positivo y negativo (con signo)

Opcionalmente, cuando sea de utilidad, al final del nombre se añadirá un número que indicará el número de bits de la señal. Así, la señal `senal_slv3`, sería un `std_logic_vector` de 3 bits (rango de 2 a 0).

Siguiendo esta notación, en el código 8.3 muestra la declaración de tres señales: `unsigned`, `signed` y `std_logic_vector`.

En las últimas sentencias del código se muestra cómo asignando los mismos bits los valores representados son distintos según el tipo de datos. En el caso de la señal de tipo `std_logic_vector`, ésta no debería de representar ningún numero, y así pasa si se utiliza el paquete `numeric_std` (código 8.1). Sin embargo, si se usa el paquete `std_logic_arith` (código 8.2) se considera que es un número sin signo, y esto puede llevar a error si no lo tienes en cuenta. Por esto no se recomienda el uso de los paquetes `std_logic_arith` y `std_logic_unsigned` (código 8.2).

```
  signal senal_us3    : unsigned (2 downto 0);
  signal senal_s3     : signed (2 downto 0);
  signal senal_slv3   : std_logic_vector (2 downto 0);
...
begin
  senal_us3  <= "111";   -- representa 7 en decimal
  senal_s3   <= "111";   -- representa -1 en decimal
  senal_slv3 <= "111";   -- no representa un numero, aunque si se usa el
                         -- paquete STD_LOGIC_UNSIGNED es 7
```

Código 8.3: Comparación entre tipos `unsigned`, `signed` y `std_logic_vector`

A continuación veremos realizar conversiones entre los tipos numéricos.

8.3. Conversión de tipos numéricos

Para convertir los tipos de datos se usan *cast* y funciones de conversión.

8.3.1. Asignación de bits

Como muestra el código 8.4, los elementos (bits) de los `unsigned`, `signed` y `std_logic_vector` se pueden asignar automáticamente entre ellos y a `std_logic`.

```
  signal senal_sl   : std_logic;
  signal senal_slv  : std_logic_vector (c_nbits-1 downto 0);
  signal senal_us   : unsigned (c_nbits-1 downto 0);
  signal senal_s    : signed (c_nbits-1 downto 0);
...
begin
  senal_sl     <= senal_us(2);  -- elemento (bit) de unsigned a std_logic
  senal_slv(0) <= senal_s(1);   -- bit de signed a bit de std_logic
  senal_us(1)  <= senal_slv(0); -- bit de std_logic_vector a bit de unsigned
  senal_s(2)   <= senal_us(1);  -- bit de unsigned a bit de signed
```

Código 8.4: Conversión directa entre elementos (bits) de vectores y con `std_logic`

8.3.2. Conversiones

Las conversiones entre tipos `unsigned` y `signed` al tipo `std_logic_vector` se debe usar un *cast*. Las señales deben tener el mismo rango.

```
  signal senal_sl   : std_logic;
  signal senal_slv  : std_logic_vector (c_bits-1 downto 0);
  signal senal_us   : unsigned (c_bits-1 downto 0);
  signal senal_s    : signed (c_bits-1 downto 0);
...
begin
  senal_slv <= std_logic_vector(senal_us);  -- de unsigned a std_logic_vector
  senal_slv <= std_logic_vector(senal_s);   -- de signed a std_logic_vector
  senal_s   <= signed (senal_slv);          -- de std_logic_vector a signed
  senal_us  <= unsigned (senal_slv);        -- de std_logic_vector a unsigned
```

Código 8.5: Conversión cast entre `signed` y `unsigned` con `std_logic_vector` (del mismo rango)

Para convertir `signed` y `unsigned` a `integer`, se deben usar funciones de conversión:

```
  signal  senal_int_ps  : integer range 0 to 255;
  signal  senal_int     : integer range -128 to 127;
  signal  senal_us8     : unsigned (7 downto 0);
  signal  senal_s8      : signed (7 downto 0);
...
begin
  senal_int_ps <= TO_INTEGER (senal_us8);  -- de unsigned a entero sin signo
  senal_int    <= TO_INTEGER (senal_s8);   -- de signed a entero con signo
  -- de entero sin signo a unsigned, 8: indica el n bits:
  senal_us8    <= TO_UNSIGNED(senal_int_ps,8);
  -- de entero con signo a signed, 8: indica el n bits:
  senal_s8     <= TO_SIGNED (senal_int,8);
```

Código 8.6: Funciones de conversión entre `signed` y `unsigned` a `integer`, usando el paquete `numeric_std`

Si se usa el paquete `std_logic_arith` las funciones de conversión son `conv_integer`, `conv_signed` y `conv_unsigned`, en vez de `to_integer`, `to_signed` y `to_unsigned` respectivamente.

La diferencia entre las asignaciones del código 8.5 con las del código 8.6 es que en las primeras se realiza un *cast* y en las segundas se utiliza una función de conversión que están en los paquetes.

Por un lado el *cast* es una reinterpretación del significado de la señal, pero la señal mantiene el mismo número de bits y el mismo formato. En el ejemplo del código 8.5, lo que antes era un vector de 8 bits sin ningún significado numérico pasa a ser un número codificado en binario con signo o sin signo según usemos el *cast* `signed` o `unsigned`. O la conversión contraria, un número binario que pasa a ser considerado un vector de 8 bits sin interpretación numérica.

Por otro lado la conversión (código 8.6) suele implicar un cambio en el formato del dato. En el ejemplo se realizan conversiones entre números escalares (`integer`) y números vectoriales (`signed` y `unsigned`).

Para convertir de `std_logic_vector` a entero se requieren dos pasos. Primero se debe realizar primero un *cast* a `signed` o `unsigned` y luego una conversión a entero. Para realizar la conversión contraria, se convierte primero el entero a `signed` o `unsigned` y luego se realiza el *cast* a `std_logic_vector`. En el código 8.7 se muestran ejemplos de estas conversiones. Se ha omitido la declaración de la mayoría de las señales, pues se corresponderían con las del código 8.6.

```
  signal  senal_slv8  : std_logic_vector (7 downto 0);
...
begin
  -- de std_logic_vector a entero sin signo:
  senal_int_ps <= TO_INTEGER(unsigned(senal_slv8));
  -- de std_logic_vector a entero con signo:
  senal_int  <= TO_INTEGER(signed(senal_slv8));
  -- de entero sin signo a std_logic_vector:
  senal_slv8 <= std_logic_vector(TO_UNSIGNED(senal_int_ps,8));
  -- de entero sin signo a std_logic_vector:
  senal_slv8 <= std_logic_vector(TO_SIGNED(senal_int,8));
```

Código 8.7: Conversión entre `integer` y `std_logic_vector`

8.4. Uso de constantes numéricas

No podremos asignar directamente una constante decimal a un `signed` o `unsigned`, aunque sí podemos comparar, o utilizarlos si interviene en una operación (por ejemplo una suma). El código 8.8 muestra una manera de realizar asignación de constantes.

```
  signal  sa_us8, sb_us8, sc_us8 :  unsigned (7 downto 0);
...
begin
  -- No se puede asignar la constante 5 directamente
  -- sa_us8 <= 5; --> esto no se puede
  sa_us8 <= TO_UNSIGNED(5,8);
  -- Las constantes se pueden usar con otras operaciones
  sb_us8 <= sa_us8 + 5;
  -- Las constantes se pueden usar con comparaciones
  sc_us8 <= TO_UNSIGNED(5,8) when (sb_us8 = 5) else TO_UNSIGNED(1,8);
```

Código 8.8: Asignación de constantes a un `unsigned` y uso de constantes en operaciones

8.5. Suma de números positivos

Para realizar una suma tenemos que considerar el rango de los operandos y el acarreo. Es interesante además que recuerdes cómo se realiza un sumador con esquemáticos [19mach].

En este apartado veremos cómo sumar números sin signo (`unsigned`), la suma de números con signo se verá en el apartado siguiente que se describe la resta.

En VHDL, la suma dos señales `unsigned` del mismo rango da como resultado un `unsigned` del mismo rango. Por consiguiente, si no lo tenemos en cuenta, perderemos el acarreo. En el código 8.9 se muestra este ejemplo.

```
  signal sa_us4, sb_us4, resul_us4 : unsigned (3 downto 0);
...
begin
  -- Suma sin acarreo:
  resul_us4 <= sa_us4 + sb_us4;
```

Código 8.9: Suma de tipos `unsigned` sin considerar el acarreo

Como la suma del código 8.9 no tiene en cuenta el acarreo nos puede llevar a resultados erróneos. Para solucionarlo tenemos que dos opciones:

- Aumentar en uno el rango del resultado
- Avisar de que ha habido un desbordamiento (*overflow*)

En el ejemplo del código 8.9, como los sumandos tienen 4 bits, el resultado de la suma tendrá como máximo 5 bits (ver figura 8.1). Esta es la regla general para cualquier número de bits, el resultado tendrá como máximo un bit más que los operandos.

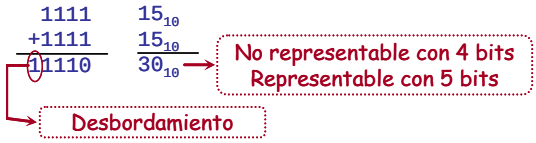


Figura 8.1: Desbordamiento en la suma de números sin signo

Sin embargo no basta con asignar el resultado a un `unsigned` de 5 bits. El código 8.10 muestra una manera **errónea** de intentar solucionar el problema.

```
  signal sa_us4, sb_us4 : unsigned (3 downto 0);
  signal resul_us5      : unsigned (4 downto 0);
...
begin
  resul_us5 <= sa_us4 + sb_us4; -- MAL
```

*Código 8.10: Suma de tipos `unsigned` considerando el acarreo de manera **errónea***

Hay que tener cuidado con descripciones como la del código 8.10 pues a veces las herramientas de diseño no la consideran errónea. Por ejemplo, la versión que usamos del ISE sólo da una advertencia (*warning*[52]) y pone directamente a cero el bit 4 del resultado. Esto haría que nuestro diseño no funcionase como esperamos. Por eso es importante leer las advertencias del ISE y simular el circuito.

Para incluir el acarreo en el resultado podemos concatenar un cero a la izquierda de los sumandos. De esta manera los sumandos tendrán 5 bits y proporcionarán un resultado de 5 bits. Para concatenar se utiliza el operador `&`.

En el código 8.11 se muestra cómo considerar los acarreos. La declaración de las señales es la misma que la del código 8.10

```
  resul_us5 <= ('0' & sa_us4) + ('0' & sb_us4);
```

Código 8.11: Suma considerando el acarreo de tipos `unsigned`

Sin embargo, no siempre podemos aumentar el tamaño del resultado. En este caso tendremos que avisar si ha habido desbordamiento. Lo podremos hacer extrayendo los cuatro primeros bits del resultado, y dejando el bit más significativo como señal de desbordamiento (`ov`).

```
signal sa_us4, sb_us4, resul_us4 : unsigned (3 downto 0);
signal resul_us5   : unsigned (4 downto 0);
signal ov          : std_logic;  -- aviso de desbordamiento (overflow)
...
begin
  resul_us5 <= ('0' & sa_us4) + ('0' & sb_us4);  -- resultado interno
  resul_us4 <= resul_us5(3 downto 0);  -- resultado de 4 bits
  ov <= resul_us5(4);                  -- senal de desbordamiento
```

Código 8.12: Suma de `unsigned` con aviso de desbordamiento

Observa en el código 8.12 que trabajar con tipos vectoriales como `unsigned` permite extraer de manera sencilla el desbordamiento (`ov`). Si utilizásemos tipos escalares sería más complicado y no tendríamos el mismo control sobre los desbordamientos y el número de bits que estamos utilizando en cada operación.

8.5.1. Informe de síntesis

Es interesante mirar los informes de la síntesis del ISE, en éstos se muestran los sumadores que se han generado y el número de bits del sumador. Para ver el informe de síntesis, dentro de la subventana *Processes*, pincha en *Synthesize - XST→View Synthesis*

[52] El *warning* que da dice así: *Width mismatch. <resul_us5> has a width of 5 bits but assigned expression is 4-bit wide.*

Report. En la ventana de la derecha aparecerá el informe, donde puedes buscar el número de sumadores/restadores que se han creado y el número de bits de cada uno de ellos. En la figura 8.2 se muestra una parte del informe para el sumador del código 8.11. Vemos que se ha creado un sumador de 5 bits, que es lo que queríamos para considerar el acarreo.

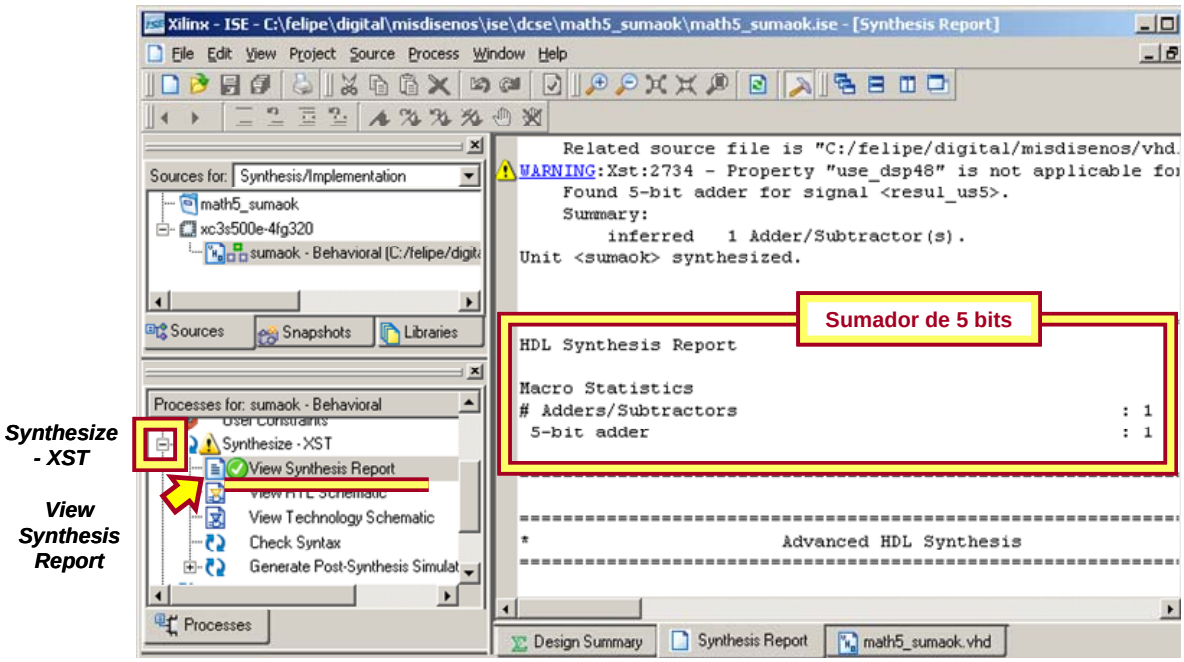


Figura 8.2: Informe de síntesis

En general es interesante echarle un vistazo al informe de síntesis, en donde se muestra más información, como el número de LUTs[53] utilizadas y estimaciones temporales.

Hay otras formas de acceder al informe de síntesis, por ejemplo accediendo al resumen del diseño[54] (*Design Summary)* y desde allí puedes pinchar en el *Synthesis Report*.

8.5.2. Esquema RTL

Una vez realizada la síntesis podemos ver el esquema RTL del circuito resultante. Para ello pinchamos en *Synthesize - XST→View RTL Schematic*. Aparecerá una caja representando al circuito con sus puertos de entrada y salida (figura 8.3).

[53] LUT: *Look-Up Table*, tablas de consulta que en las FPGAs se utilizan para implementar las funciones lógicas

[54] Recuerda los pasos 1 y 2 de la figura 7.35

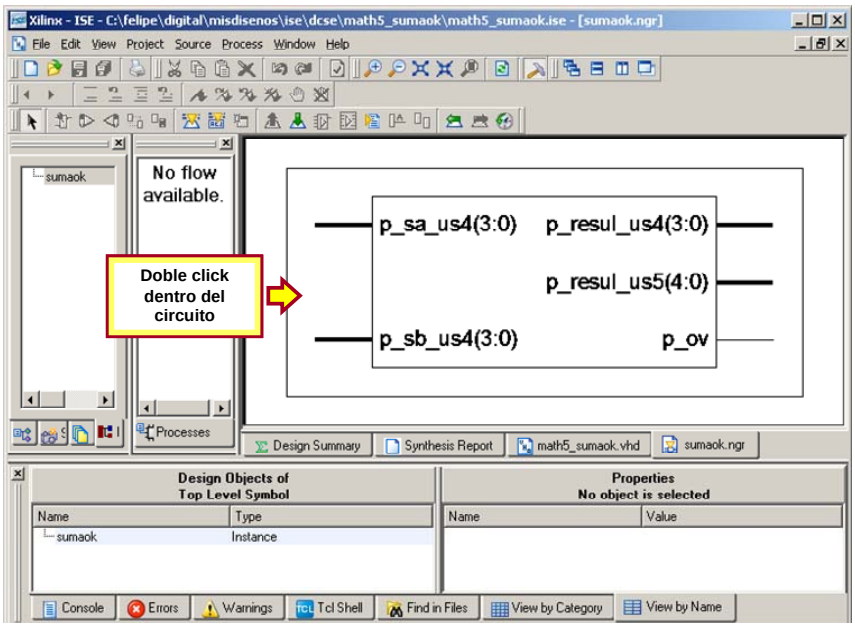


Figura 8.3: Esquema RTL

Haciendo doble click dentro de la caja veremos el esquema del sumador (figura 8.4).

Pinchando en los elementos del esquema se muestra información en las ventanas de abajo. De esta manera se pueden identificar las señales y ver cómo se ha sintetizado nuestro diseño. Para circuitos muy grandes se requiere paciencia por la gran cantidad de elementos que puede llegar a haber.

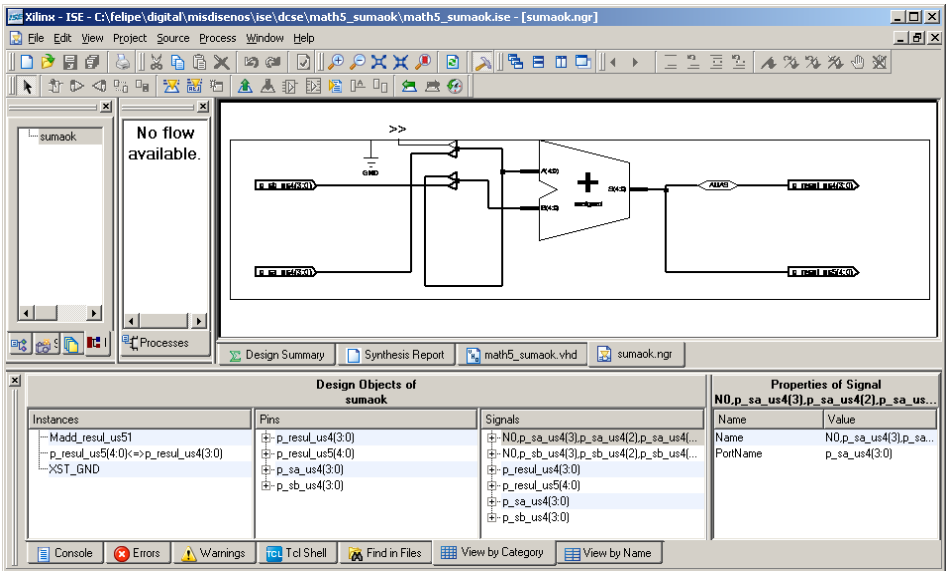


Figura 8.4: Esquema RTL del sumador

En circuitos estructurales se podrá pinchar en las cajas que habrá dentro de cada circuito. Para volver a los circuitos de mayor jerarquía basta con hacer doble click fuera del circuito.

8.6. Resta de números positivos

En este apartado veremos cómo hacer la resta de números positivos, considerando el resultado como número positivo (magnitud o módulo) y teniendo una señal de un bit que indicará el signo. Esto es equivalente a la representación en signo y magnitud.

Si queremos realizar una resta de números positivos y queremos obtener el resultado como número positivo, tenemos que poner como minuendo el número mayor y como sustraendo el menor. Por lo tanto, antes de realizar la resta tendremos que comparar y realizar las asignaciones correspondientes según quién sea el mayor. Si hemos tenido que intercambiar minuendo por sustraendo deberemos avisar que el resultado es negativo.

Este circuito se podría describir como muestra el código 8.13. Como ya sabemos, existen muchas alternativas equivalentes para describir este circuito, por ejemplo, usando procesos, otras señales intermedias,...

```
entity restapositiva is
 Port (
   p_minuen_us    : in  unsigned (3 downto 0);  -- Minuendo
   p_sustr_us     : in  unsigned (3 downto 0);  -- Sustraendo
   p_resta_us     : out unsigned (3 downto 0);  -- Resta
   -- si es 1 indica que el resultado es negativo minuendo < sustraendo
   p_dif_negativa : out std_logic
  );
end restapositiva;

architecture behavioral of restapositiva is
  signal dif_negativa     : std_logic;
begin
  -- si el sustraendo es mayor que el minuendo la diferencia sera negativa
  dif_negativa <= '0' when (p_minuen_us >= p_sustr_us) else '1';

  p_resta_us <= (p_minuen_us - p_sustr_us) when dif_negativa = '0' else
                (p_sustr_us - p_minuen_us);

  p_dif_negativa <= dif_negativa;
end behavioral;
```

Código 8.13: Resta de números positivos

Como la resta de números positivos en la que el minuendo es el mayor está acotada entre el valor del minuendo y cero, en este caso no tendremos desbordamiento.

8.7. Suma y resta de números enteros

Para sumar y restar números enteros podemos combinar los apartados anteriores de modo que siempre sumemos o restemos números positivos y calculemos el signo del resultado. Para esto es necesario realizar unas sencillas comprobaciones de los signos y las magnitudes de los operandos. Esta sería la manera de operar con números representados con signo y magnitud.

Otra alternativa, más utilizada en los circuitos digitales, es operar en complemento a dos. El complemento a dos permite operar números positivos y negativos sin preocuparnos si tienen distinto signo. Así que evitamos hacer las comprobaciones que se hacen con signo y magnitud. En el apartado 8.7.3 se ha incluido un pequeño repaso de los números en complemento a dos. En la referencia [19mach] se explica cómo se realiza un sumador/restador de números en complemento a dos usando esquemáticos.

En VHDL, los números en complemento a dos los podemos declarar como un `signed`. Para realizar la suma o la resta sólo nos tenemos que preocupar de los rangos y del

desbordamiento. Los desbordamientos en las operaciones en complemento a dos son distintos que los producidos con números sin signo (recuerda la figura 8.1).

En la suma, el desbordamiento se produce cuando los dos sumandos tienen el mismo signo y el resultado tiene un signo distinto. Esto está explicado en la figura 8.5 del apartado 8.7.3.

Con esto ya tenemos la información necesaria para describir el sumador de números enteros. El código 8.14 muestra una posible implementación del circuito, que incluye el aviso de desbordamiento.

```
entity sumaentera is
 Port (
   p_sum1_s     : in  signed (3 downto 0);
   p_sum2_s     : in  signed (3 downto 0);
   p_resul_s    : out signed (3 downto 0);
   p_ov  : out std_logic  -- p_ov='1' -> desbordamiento
  );
end sumaentera;

architecture behavioral of sumaentera is
  signal resul_s     : signed (3 downto 0);
begin
  resul_s <= p_sum1_s + p_sum2_s;

  -- calculo del desbordamiento: resultado con distinto signo que los 2 sumandos
  p_ov <= (    p_sum1_s(3)  and     p_sum2_s(3)  and not(resul_s(3))) or
          (not(p_sum1_s(3)) and not(p_sum2_s(3)) and     resul_s(3) );

  p_resul_s <= resul_s;
end behavioral;
```

Código 8.14: Suma de números enteros

En complemento a dos, la resta no es más que una suma en la que se cambia el signo del sustraendo[55]. Para describir una resta en VHDL, simplemente ponemos la operación de la resta y calculamos el desbordamiento consecuentemente, es decir, negando el sustraendo.. El código 8.15 muestra las sentencias que cambiarían respecto a la suma (código 8.14).

```
...
  resul_s <= p_sum1_s - p_sum2_s;

  -- calculo del desbordamiento: resultado con distinto signo que los 2 sumandos
  p_ov <= (    p_sum1_s(3)  and not(p_sum2_s(3)) and not(resul_s(3))) or
          (not(p_sum1_s(3)) and     p_sum2_s(3)  and     resul_s(3) );
```

Código 8.15: Resta de números enteros

8.7.1. Aumento del rango de la suma en complemento a dos

La suma[56] de números enteros puede desbordarse (figura 8.5). Igual que en el caso de la suma de números positivos (apartado 8.5), si no queremos tener desbordamiento tenemos que aumentar el rango del resultado en un bit. En el código 8.11 vimos cómo resolver esta situación para los números positivos. En esta solución concatenábamos un cero a la izquierda de cada sumando.

Sin embargo, con números enteros en complemento a dos no podemos hacer lo mismo porque estaríamos cambiando el signo de los números negativos. Por ejemplo, si tenemos el número `1101` (-3) y le concatenamos un cero a la izquierda lo convertiremos en `01101`, que no sólo ahora es un número positivo sino que lo hemos convertido en el número 13.

[55] Recuerda que en la representación en complemento a dos, cambiar el signo de un número es calcular su complemento a dos

[56] Y la resta, que como ya hemos dicho no es más que una suma cambiando el signo del sustraendo

La solución es concatenar un cero a la izquierda si el número es positivo y un uno si es el número es negativo. Es decir, concatenar con el valor del bit más significativo, que es el bit que indica el signo. Así, por ejemplo, si tenemos `0010` (2), obtendremos `00010`, que sigue siendo el 2. Y si tenemos `1110` (-2), obtendremos `11110`, que sigue siendo el -2.

Por lo tanto, la adaptación del código 8.11 a números enteros se muestra en el código 8.16.

```
resul_s5 <= (sa_s4(3) & sum1_s4) + (sb_s4(3) & sum2_s4);
```

Código 8.16: Suma de tipos `signed` considerando el acarreo

8.7.2. Aumento del rango usando funciones y atributos *

Una alternativa al código 8.16 es usar la función `resize`. La función `resize` se encarga de redimensionar los vectores `signed` y `unsigned`. Esta función está en el paquete `numeric_std`, por lo que no la podrás utilizar si estas usando los paquetes `std_logic_arith` y `std_logic_unsigned` (recuerda el apartado 8.1).

Con esta función el código 8.16 quedaría como muestra el código 8.17.

```
resul_s5 <= resize(sum1_s4, 5) + resize(sum2_s4, 5)
```

Código 8.17: Suma de tipos `signed` considerando el acarreo y utilizando la función `resize`

En el primer argumento de la función `resize` se pone el vector `signed` o `unsigned`, y en el segundo argumento se pone el número de bits a los que se quiere redimensionar. La función `resize` redimensionará el vector adecuadamente dependiendo del tipo y del signo del vector.

Sin embargo podemos observar que realmente no hay mucha diferencia entre los códigos 8.17 y 8.16. Aunque en caso que el cambio sea de muchos bits el código 8.17 será más claro y conciso. Por otro lado, `resize` es tanto para `signed` como para `unsigned`. Si lo hacemos nosotros, tendremos que elegir entre el código 8.16 y el código 8.10, según sea `signed` o `unsigned`.

Pero el código 8.17 se puede optimizar, en vez de usar directamente el número de bits a los que queremos convertir los sumandos (5 en el ejemplo), podemos usar el atributo `length`, que nos devuelve el número de bits (longitud) de una señal. Así, la expresión `resul_s5'length` nos devolverá el número de bits de la señal `resul_s5`, que es 5.

Aunque no los hemos nombrado hasta ahora, los atributos en VHDL se utilizan bastante. Por ejemplo, nosotros ya hemos utilizado el atributo `event` para la expresar el evento de la señal de reloj: `clk'event` (recuerda por ejemplo el apartado 2.6).

Los atributos se ponen con un apóstrofe a continuación de la señal. Hay muchos tipos de atributos, pero no todos son sintetizables o no son aceptados por las herramientas de diseño. Así que antes de utilizarlos debes comprobar que no hay problema con ellos.

Utilizando el atributo `length`, el código 8.17 quedará:

```
resul_s5 <= resize(sum1_s4, resul_s5'length) + resize(sum2_s4, resul_s5'length);
```

Código 8.18: Suma de tipos `signed` considerando el acarreo y utilizando la función `resize` con el atributo `length`

Describiendo el sumador de esta manera no lo hacemos depender de un valor numérico, como lo era el 3 en el código 8.16 ó el 5 en el código 8.17. Esto tiene muchas ventajas, ya que si queremos cambiar el número de bits del sumador ya sea para reutilizarlo o porque hayan cambiado las especificaciones, no necesitamos cambiar el valor numérico, sino que

se hace automáticamente al cambiar el rango de las señales en su declaración. Esto nos podrá ahorrar errores de edición especialmente en circuitos grandes.

8.7.3. Breve repaso del complemento a dos *

En este apartado se hace un pequeño recordatorio de la representación de números en complemento a dos. En la tabla 8.1 se muestra la equivalencia de los números en compelemento a dos de cuatro bits.

decimal	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
C'2	1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111

Tabla 8.1: Números en complemento a dos de 4 bits y su correspondiente decimal

Los números positivos (y el cero) representados en complemento a dos siempre tienen un cero en el bit más significativo. Como se ve en la tabla 8.1 los números positivos en complemento a dos se corresponden con los binarios puros.

Los números negativos tienen un uno en el bit más significativo. Para saber con qué número se corresponden se invierten los ceros por unos y los unos por ceros y al resultado se le suma uno.

Por ejemplo, si tenemos el `1011`.

- Vemos que es un número negativo porque tiene un uno en el bit más significativo.
- Invertimos ceros por unos y unos por ceros: `0100`
- Sumamos uno (`0100+0001) = 0101`, que es el número 5 en binario
- Por lo tanto, `1011` es el número -5

Esta operación es la misma para hallar el complemento a dos de cualquier número (negarlo).

En la suma de números en complemento a dos el desbordamiento se produce cuando los dos sumandos tienen el mismo signo y el resultado tiene un signo distinto. La figura 8.5 muestra con ejemplos cuándo se produce desbordamiento.

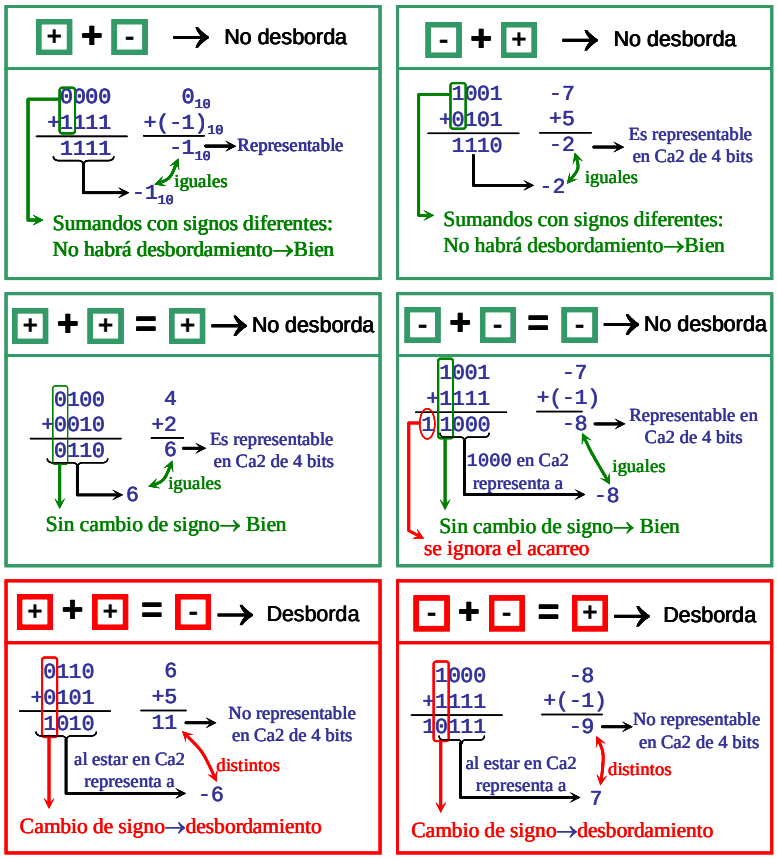


Figura 8.5: Ejemplos de los casos de desbordamiento en la suma en complemento a dos

Para la resta, es equivalente que la suma pero habiendo cambiado previamente el signo del sustraendo.

8.8. Multiplicación

El sintetizador del ISE implementa la multiplicación de dos números enteros. El resultado de la multiplicación tendrá tantos bits como la suma del número de bits de los productos. El código 8.19 muestra cómo se describe una multiplicación de números enteros sin signo.

```
...
  signal fact1_us4, fact2_us4 : unsigned (3 downto 0);
  signal prod_us8             : unsigned (7 downto 0);
begin
  prod_us8 <= fact1_us4 * fact2_us4;
...
```

Código 8.19: Producto de dos `unsigned` produce un `unsigned` con un número de bits resultado de la suma del número de bits de los factores

Las FPGA que usamos llevan integrados multiplicadores, por tanto, las operaciones de multiplicación entera quedan eficientemente implementadas automáticamente. Sin embargo, puede ser que necesitemos implementar más multiplicadores que los que tiene la FPGA. La FPGA de la *XUPV2P* tiene 136, la de la *Nexys2* tiene 20 y la de la *Basys* tiene 4. Esto lo puedes ver en el informe de síntesis, que indica el número de multiplicadores totales y usados. En el caso en el que se necesiten más multiplicadores que los que tiene la FPGA, el sintetizador los implementará con lógica combinacional.

En el caso en el que uno de los factores de la multiplicación sea una constante, ésta podrá ser más simple. Especialmente si la constante es potencia de dos. El mecanismo de la

multiplicación por dos en binario es como la multiplicación por diez en decimal, es decir, basta con añadir ceros a la derecha. Así que una multiplicación por dos es un desplazamiento a la izquierda, con esto se reduce muchísimo la lógica requerida para su implementación.

Cuando se multiplica por una constante, el rango del resultado es el doble del número de bits del factor que no es constante. Este lleva implícito una consecuencia que hay que tener en cuenta: no podemos multiplicar un número por una constante de rango mayor que el número.

Así, el código 8.20 sería como multiplicar por cero, porque se trunca la constante al mismo número de bits que `fact_us4`. Por tanto el 16 (`10000b`) se quedaría en 0 (`0000b`) por haber truncado el uno que está en el bit más significativo. Para solucionarlo, tenemos que ampliar el rango del factor en un bit y ampliar el rango del producto. En el código 8.21 se muestra cómo se ha hecho esto.

```
...
 signal fact_us4: unsigned(3 downto 0);
 signal prod_us8: unsigned(7 downto 0);
begin
  prod_us8 <= fact_us4 * 16;
...
```

*Código 8.20: Producto por una constante de rango mayor. **Erróneo***

```
...
 signal fact_us4: unsigned(3 downto 0);
 signal prod_us10:unsigned(9 downto 0);
begin
  prod_us10 <= ('0'&fact_us4) * 16;
...
```

*Código 8.21: Producto por una constante de rango mayor. **Correcto***

8.8.1. Algoritmo de la multiplicación

En este apartado se explicará cómo es el algoritmo de la multiplicación y en los siguientes se describirán varias maneras de implementar un multiplicador. Aunque un multiplicador lo podemos describir como se mostró en el código 8.19 y el sintetizador se encarga de implementarlo, este apartado puede ser útil porque se darán algunas indicaciones útiles para implementar otros operadores como la división, se analizarán distintas estrategias para implementar ciertos tipos de operadores y se verán maneras más avanzadas de describir circuitos en VHDL.

Antes ver cómo implementar la multiplicación vamos a analizar cuál es el mecanismo que utilizamos al multiplicar. La figura 8.6 muestra una multiplicación con números decimales y a su derecha la misma multiplicación en binario.

```
   9
 x 13
 ----
  27
+ 9
 ----
 117
```

```
    1001 :   9
  x 1101 :  13
 ------    ---
    1001     9
   0000      0
  1001      36
+ 1001      72
 -------   ---
 1110101 : 117
```

Figura 8.6: Ejemplos de multiplicación entera en decimal y en binario

Si observamos la figura 8.6 podemos ver que los mecanismos de la multiplicación decimal y binaria son similares: se multiplica cada cifra del factor de abajo por el factor de arriba. El resultado de la multiplicación se desplaza tantas cifras a la izquierda como la cifra del factor de abajo. Recuerda que desplazar a la izquierda es equivalente a multiplicar por la base, en decimal 10 y en binario 2. Por último se suman todas las multiplicaciones parciales y se obtiene el producto.

En el caso de la multiplicación binaria las multiplicaciones parciales son muy sencillas porque las únicas posibilidades son multiplicar por cero o por uno. Como veremos, estas multiplicaciones se pueden implementar con puertas and.

8.8.2. Implementación de un multiplicador combinacional

Como hemos visto, para implementar la multiplicación tendremos que realizar sumas, desplazamientos y multiplicaciones por uno y cero.

El diseño en esquemáticos de este circuito se podría implementar como muestra la figura 8.7. El multiplicador tendrá tantas etapas como bits de los factores. La primera etapa es la única que no tiene sumador[57].

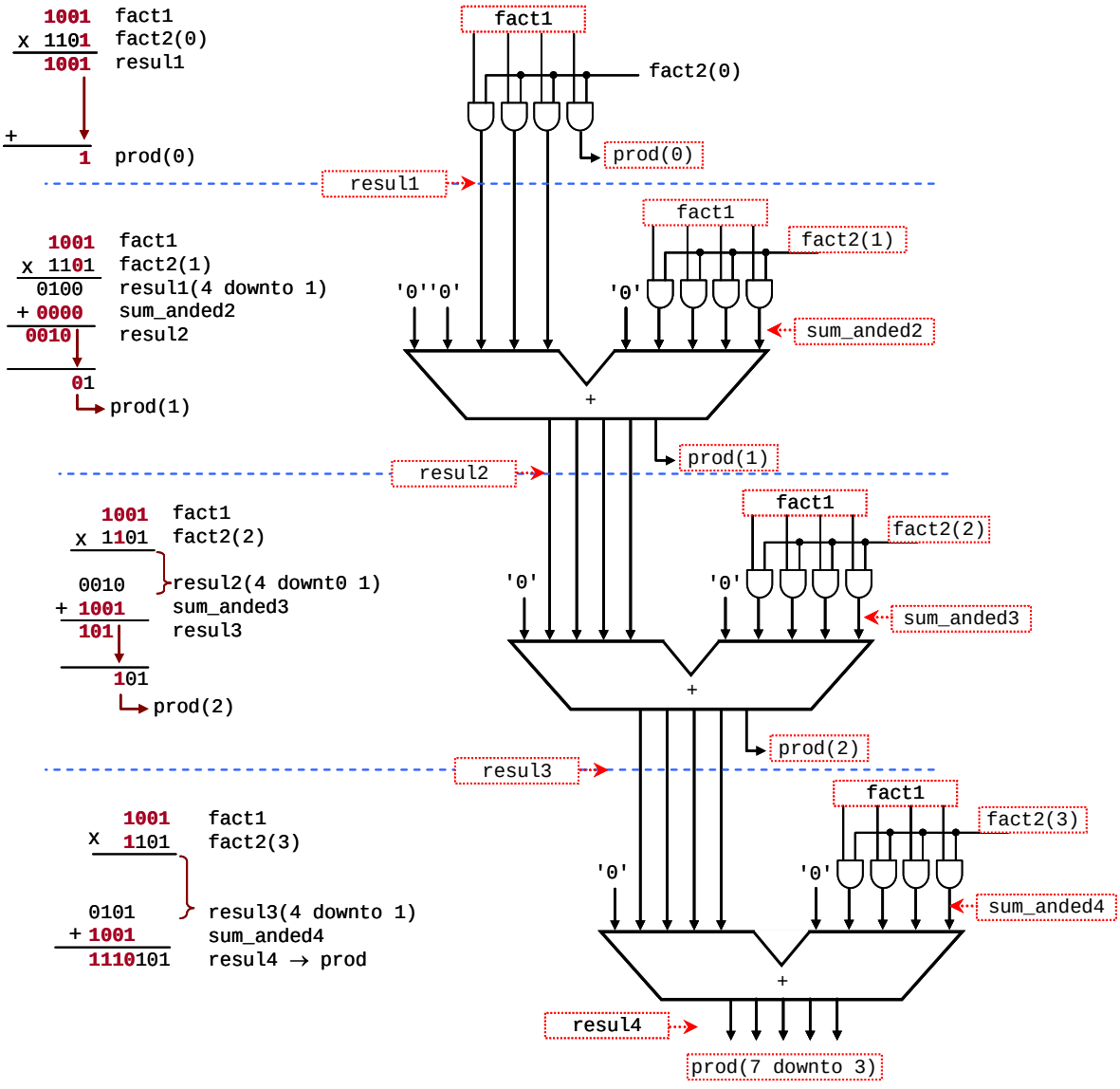


Figura 8.7: Esquema del multiplicador combinacional

En la implementación se han hecho las sumas parciales conforme se va multiplicando. La figura 8.8 muestra la correspondencia de los nombres que se han dado a las señales.

[57] También se puede poner un sumador en el que uno de los sumandos es cero.

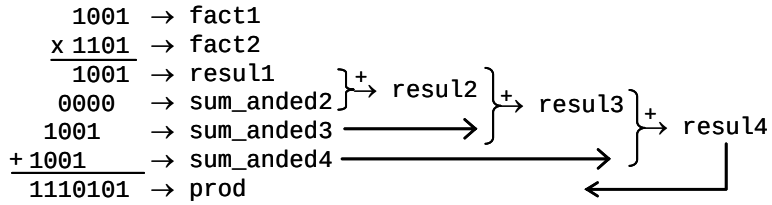


Figura 8.8: Sumas parciales para la multiplicación

Las señales son:

- Los números que se van a multiplicar (factores): `fact1` y `fact2`
- Los sumandos que ya se han multiplicado usando una puerta and: `sum_anded1`, `sum_anded2`, `sum_anded3`
- Las sumas parciales: `resul1`, `resul2`, `resul3` y `resul4`. Las señales `sum_anded1` y `resul1` serían equivalentes
- El resultado de la multiplicación (producto): `prod`

El VHDL de este esquemático podría ser el mostrado en el código 8.22.

```
entity mult_comb is
 Port (
   fact1   : in  unsigned (3 downto 0);
   fact2   : in  unsigned (3 downto 0);
   prod    : out unsigned (7 downto 0)
  );
end mult_comb;

architecture behavioral of mult_comb is
  signal sum_anded1, sum_anded2, sum_anded3, sum_anded4 : unsigned(3 downto 0);
  signal resul1,     resul2,     resul3,     resul4     : unsigned(4 downto 0);
begin
-- primera etapa
  sum_anded1 <= fact1 when fact2(0) = '1' else (others => '0');
  resul1     <= '0' & sum_anded1;
  prod(0)    <= resul1(0);
-- segunda etapa
  sum_anded2 <= fact1 when fact2(1) = '1' else (others => '0');
  resul2     <= ('0' & resul1(4 downto 1)) + ('0' & sum_anded2);
  prod(1)    <= resul2(0);
-- tercera etapa
  sum_anded3 <= fact1 when fact2(2) = '1' else (others => '0');
  resul3     <= ('0' & resul2(4 downto 1)) + ('0' & sum_anded3);
  prod(2)    <= resul3(0);
-- cuarta etapa
  sum_anded4 <= fact1 when fact2(3) = '1' else (others => '0');
  resul4     <= ('0' & resul3(4 downto 1)) + ('0' & sum_anded4);
-- final:
  prod(7 downto 3) <= resul4;
end behavioral;
```

Código 8.22: Multiplicación combinacional con sentencias concurrentes

Fíjate que las sentencias del código 8.22 son concurrentes, es decir, no están dentro de un proceso.

8.8.3. Implementación de un multiplicador combinacional genérico *

El multiplicador del apartado anterior es de cuatro bits[58], siendo el resultado de ocho bits. Podemos ver que excepto la primera etapa, el resto de etapas son muy similares. Si ahora tuviésemos que diseñar un multiplicador de 16 bits (32 bits para el producto) tendríamos que implementar las 16 etapas copiando y pegando de las anteriores. Esto, además del tiempo invertido, favorece la aparición de errores.

[58] También se llama multiplicador 4x4

Para solucionar esto el VHDL permite generar sentencias repetitivas mediante la sentencia `generate`. En este apartado veremos primero la sentencia `generate` que nos permite replicar sentencias concurrentes similares y posteriormente la veremos aplicada a un multiplicador genérico en el que podremos variar el número de bits de los factores con genéricos (recuerda el apartado 6.2).

8.8.3.1. La sentencia `generate` *

En VHDL se pueden generar sentencias repetitivas mediante la sentencia **`generate`**. Por ejemplo, si queremos asignar a una señal los elementos de otra señal pero invertidos, podríamos hacerlo como en el código 8.23. Como siempre, el VHDL tiene varias alternativas para describir la misma cosa, y esta sentencia `generate` no es la única manera de describir esta funcionalidad. En el código 8.24 se muestra cómo hacerlo.

```
...
begin
  p_out(0) <= p_in(3);
  p_out(1) <= p_in(2);
  p_out(2) <= p_in(1);
  p_out(3) <= p_in(0);
end behavioral;
```

Código 8.23: Asignación invirtiendo el orden de los bits

```
entity gen is
 Port (
  p_in  : in std_logic_vector (3 downto 0);
  p_out : out std_logic_vector (3 downto 0)
 );
end gen;

architecture behavioral of gen is
begin
  GENER: for i in 0 to 3 generate
    p_out(i) <= p_in(3-i);
  end generate;
end behavioral;
```

Código 8.24: Asignación invirtiendo el orden de los bits usando la sentencia `generate`

Las características básicas de la sentencia `generate` son:

- Es una sentencia concurrente, por lo que no puede ir dentro de un proceso[59].
- Replica las sentencias que contiene conforme al índice del `for`[60] (en el ejemplo: `i`)
- Este índice no se declara y el rango no puede ser variable (pero puede ser un genérico).
- La sentencia `generate` empieza con una etiqueta (en el ejemplo: `GENER`), similar a las etiquetas de los procesos y la referencia a componentes.

En el ejemplo puedes observar que si copias la sentencia interna del `generate` para los cuatro valores del índice se generan las mismas sentencias del código 8.23.

8.8.3.2. Multiplicación con sentencia `generate` *

Ahora que conocemos la sentencia `generate` podemos reescribir el multiplicador (código 8.22) de modo que el ancho de los factores sea genérico. En el código 8.25 se muestra el multiplicador combinacional genérico.

[59] Dentro de los procesos se puede utilizar la sentencia `loop` que tiene algunas similitudes

[60] El `generate` también puede ser condicional con un `if`, en vez de iterativo con el `for`.

```
entity mult_comb_gen is
 Generic (
   g_bits   : natural := 4
 );
 Port (
   fact1   : in  unsigned (g_bits-1 downto 0);
   fact2   : in  unsigned (g_bits-1 downto 0);
   prod    : out unsigned (2*g_bits-1 downto 0)  -- doble de bits para el producto
  );
end mult_comb_gen;

architecture behavioral of mult_comb_gen is
  type t_vector_sumand is array (1 to g_bits) of unsigned (g_bits-1 downto 0);
  type t_vector_resul  is array (1 to g_bits) of unsigned (g_bits downto 0);
  signal sum_anded : t_vector_sumand;
  signal resul     : t_vector_resul;
begin
-- primera etapa
  sum_anded(1) <= fact1 when fact2(0) = '1' else (others => '0');
  resul(1)     <= '0' & sum_anded(1);
  prod(0)    <= resul(1)(0);
-- resto de etapas
  gen_mult: for i in 1 to g_bits-1 generate
    sum_anded(i+1) <= fact1 when fact2(i)='1' else (others =>'0');
    resul(i+1) <= ('0' & resul(i)(g_bits downto 1)) + ('0' & sum_anded(i+1));
    prod(i)    <= resul(i+1)(0);
  end generate;
-- final:
  prod(2*g_bits-1 downto g_bits) <= resul(g_bits)(g_bits downto 1);
end behavioral;
```

Código 8.25: Multiplicador combinacional genérico

Observando el código 8.25 podemos ver que:

- El número de bits de los factores viene definido por el genérico `g_bits`
- Las señales `sum_anded1`, `sum_anded2`, ... y `resul1`, `resul2`, ... se ha tenido que agrupar en vectores para poder usarlas de manera iterativa con el índice del `generate`.
- Para poder declarar estos vectores hay que definir un tipo de datos que es un vector de vectores. En este caso son vectores de `unsigned`. Luego las señales se declaran con estos nuevos tipos.

En general no se recomienda realizar el diseño genérico a la primera, sino realizar una primera versión para un caso particular, ver que funciona bien y luego generalizarla. Ya que el diseño genérico es más complicado.

*8.8.3.3. Banco de pruebas para el multiplicador genérico **

Al hacer un diseño genérico como el que acabamos de ver, tenemos que realizar un banco de pruebas que se adapte a los distintos valores del diseño. En el código 8.26 se muestra un ejemplo de este tipo de bancos de prueba.

```
entity tb_mult_comb_gen is
 Generic (
   g_bits   : natural := 4
 );
end tb_mult_comb_gen;

architecture TB of tb_mult_comb_gen is
  component mult_comb_gen
    Generic (
      g_bits   : natural := 4
    );
    Port (
      fact1   : in  unsigned (g_bits-1 downto 0);
      fact2   : in  unsigned (g_bits-1 downto 0);
      prod    : out unsigned (2*g_bits-1 downto 0)
    );
  end component;

  signal fact1, fact2 : unsigned (g_bits-1 downto 0);
  signal prod         : unsigned (2*g_bits-1 downto 0);
begin
  uut: mult_comb_gen
  generic map (
    g_bits  => g_bits
  )
  port map(
    fact1  => fact1,
    fact2  => fact2,
    prod   => prod
  );

  P_stim: Process
  begin
    for i in 0 to 2**g_bits-1 loop
      fact1 <= to_unsigned (i, g_bits);
      for j in 0 to 2**g_bits-1 loop
        fact2 <= to_unsigned (j, g_bits);
        wait for 50 ns;
        assert prod = i*j
           report "Fallo en la multiplicacion"
           severity ERROR;
        wait for 50 ns;
      end loop;
    end loop;
    wait;
  end process;
end TB;
```

Código 8.26: Banco de pruebas para el multiplicador combinacional genérico

Fíjate que este banco de prueba es exhaustivo porque realiza todas las multiplicaciones posibles para el rango de los factores. El banco de pruebas incluye además la sentencia `assert` que hace que autocompruebe si ha habido algún error.

Observa además que en los rangos a veces se utiliza la multiplicación (un sólo asterisco) y otras la exponenciación (dos asteriscos), asegúrate que entiendes por qué.

8.8.4. Implementación de un multiplicador estructural *

Como ya sabemos, el VHDL permite múltiples descripciones para la misma funcionalidad. Una alternativa a la descripción del multiplicador combinacional del apartado 8.8.2 es implementarlo de manera estructural.

Por ejemplo, podemos describir la estructura que se repite en la figura 8.7 como un componente. Es decir, describir la entidad de la figura 8.9

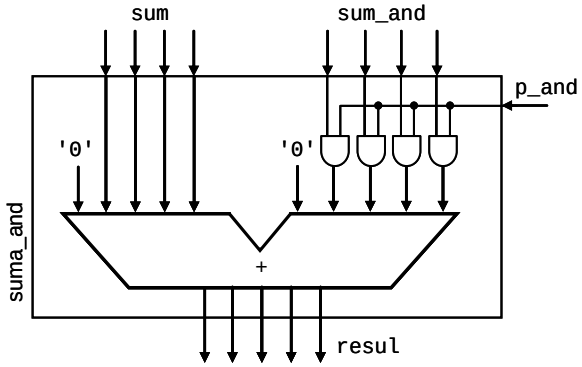


Figura 8.9: Componente básico del multiplicador

La descripción del módulo de la figura 8.9 no tiene mayor dificultad y se deja como ejercicio. El módulo estructural tampoco es difícil, se ha incluido en el código 8.27. Observa que a diferencia de las versiones de los códigos 8.22 y 8.25, en la primera etapa se ha puesto un sumador con un sumando igual a cero, por lo que funcionalmente son equivalentes.

```
entity mult_struct is
 port (
   fact1   : in  unsigned (3 downto 0);
   fact2   : in  unsigned (3 downto 0);
   prod    : out unsigned (7 downto 0)
  );
end mult_struct;

architecture behavioral of mult_struct is
  constant c_cero : unsigned (3 downto 0) := "0000";
  signal resul0, resul1, resul2, resul3   : unsigned (4 downto 0);

  component suma_and is
    port (
      sum     : in  unsigned (3 downto 0);
      sum_and : in  unsigned (3 downto 0);
      p_and   : in  std_logic;
      resul   : out unsigned (4 downto 0)
    );
  end component;

begin
```

```
  C_SUM_AND0: suma_and
    port map (
      sum     => c_cero,
      sum_and => fact1,
      p_and   => fact2(0),
      resul   => resul0
    );
  prod(0) <= resul0(0);

  C_SUM_AND1: suma_and
    port map (
      sum     => resul0 (4 downto 1),
      sum_and => fact1,
      p_and   => fact2(1),
      resul   => resul1
    );
  prod(1) <= resul1(0);

  C_SUM_AND2: suma_and
    port map (
      sum     => resul1 (4 downto 1),
      sum_and => fact1,
      p_and   => fact2(2),
      resul   => resul2
    );
  prod(2) <= resul2(0);

  C_SUM_AND3: suma_and
    port map (
      sum     => resul2 (4 downto 1),
      sum_and => fact1,
      p_and   => fact2(3),
      resul   => resul3
    );
  prod(7 downto 3) <= resul3;
end behavioral;
```

Código 8.27: Multiplicador estructural

8.8.5. Implementación del multiplicador estructural genérico *

Dentro de las sentencias `generate` se pueden referenciar componentes y por lo tanto el multiplicador estructural del código 8.27 se puede hacer genérico.

```
architecture behavioral of mult_struct is
  type   t_vector_resul is array (0 to g_bits) of unsigned (g_bits downto 0);
  signal resul   : t_vector_resul;

  component suma_and is
    generic (
      g_bits   : natural := 4
    );
    port (
      sum     : in  unsigned (g_bits-1 downto 0);
      sum_and : in  unsigned (g_bits-1 downto 0);
      p_and   : in  std_logic;
      resul   : out unsigned (g_bits downto 0)
    );
  end component;

begin
```

```
  resul(0) <= (others => '0');

  GEN_SUMA: for i in 0 to g_bits-1 generate

    C_SUM_AND: suma_and
      generic map (
        g_bits  => g_bits
      )
      port map (
        sum     => resul(i)(g_bits downto 1),
        sum_and => fact1,
        p_and   => fact2(i),
        resul   => resul(i+1)
      );

      prod(i) <= resul(i+1)(0);
  end generate;

  prod(2*g_bits-1 downto g_bits) <= resul(g_bits)(g_bits downto 1);
end behavioral;
```

Código 8.28: Multiplicador estructural genérico

La entidad del multiplicador genérico no se ha incluido porque es similar a la del multiplicador combinacional genérico (código 8.25).

Hay ciertas cosas interesantes del código 8.27:

- Dentro del `generate` hay una referencia a componente y una sentencia concurrente.
- Como la primera etapa se ha incluido en el `generate` (a diferencia del código 8.25), el tipo `t_vector_resul` tiene un elemento más. Este elemento es el del índice cero y su valor es todo cero. Este elemento es el que se suma en la primera etapa, y es equivalente a no realizar la suma.
- Por la razón anterior, el `for` del `generate` tiene una iteración más

El banco de pruebas de este multiplicador puede ser similar al del multiplicador combinacional genérico del código 8.26.

8.9. Diseño considerando las prestaciones

El objetivo de este apartado es introducirnos en el análisis de las prestaciones de un circuito de procesamiento y aprender distintas técnicas para modificar las prestaciones. En el siguiente subapartado veremos un resumen de las prestaciones y cómo se deben considerar en el diseño. Posteriormente mediante el operador de multiplicación aprenderemos de manera práctica a utilizar algunas de estas técnicas.

8.9.1. Análisis de las prestaciones

Entre las prestaciones de un circuito se incluyen la velocidad de procesamiento y el área[61]. La velocidad de procesamiento depende de muchos factores, como son la frecuencia de reloj a la que funciona el circuito, la velocidad de propagación de las señales y la tecnología empleada. El área es la cantidad de lógica empleada para implementar un circuito.

Para implementar un circuito el diseñador tiene ciertas libertades y restricciones. Por ejemplo, en nuestro caso podríamos usar la placa XUP, la Nexys2 o la Basys, esto nos

[61] El consumo energético también es un componente muy importante en las prestaciones de un circuito. Un circuito con bajo consumo energético necesitará menos energía (menor tamaño de la batería o mayor duración), disipará menos calor y será más fiable porque no se calentará tanto. El análisis del consumo está fuera de los objetivos de este libro

daría ciertas libertades, pues si nuestro diseño fuese muy grande, podríamos utilizar la XUP ya que su FPGA (*Virtex2p*) contiene un mayor número de celdas lógicas y multiplicadores. Evidentemente esto supondría un mayor coste de dinero y tener que usar una placa de mayor tamaño.

Otro ejemplo sería la frecuencia de reloj, la XUP tiene un reloj de 100 MHz mientras que la Nexys2 tiene un reloj de 50 MHz. Por otro lado la Basys tiene un reloj configurable mediante un *jumper* de 100, 50 ó 25 MHz, lo que nos daría más libertad. Sin embargo, las FPGAs *Spartan3e* de la Nexys2 y la Basys son más lentas que la *Virtex2p*. En otros diseños el diseñador podría tener la libertad de elegir la frecuencia de funcionamiento cambiando el circuito del reloj.

Lo que acabamos de decir nos podría llevar a pensar que el diseñador está limitado por circunstancias externas (frecuencia de reloj, FPGA utilizada, ...) y que el tipo de diseño VHDL que haga no tiene ninguna influencia en sus prestaciones. Afortunadamente esto no es así y una gran parte de las prestaciones del circuito viene determinada por la descripción del circuito por parte del diseñador.

Una de las cosas que más influyen en las prestaciones es el algoritmo empleado. Por ejemplo, en vez del algoritmo de multiplicación escogido en el apartado 8.8.1 podíamos haber implementado la multiplicación sumando un factor tantas veces como el valor del otro factor. Es decir, si tenemos que multiplicar 7 x 9, podemos calcular el producto sumando siete veces el número nueve o sumando nueve veces el número siete. Este algoritmo es mucho menos eficiente que el que hemos visto en el apartado 8.8.1 ya que necesita más sumadores (área) y más tiempo para completarse. Puedes hacer la prueba para comprobarlo.

Así que la elección del algoritmo es un aspecto clave en las prestaciones de un circuito que incluso podría hacer que un diseño funcione más eficientemente en una tecnología más barata que otro diseño con peor algoritmo en una tecnología con más prestaciones.

Pero la elección del algoritmo no es la única opción con la que cuenta el diseñador. Una vez escogido un algoritmo, el diseñador puede hacer variaciones para ajustarse a sus especificaciones y restricciones. Normalmente, una vez elegido un algoritmo se puede elegir en mejorar una prestación a costa de la otra. Es decir, podemos emplear más área para hacer el circuito más rápido, y lo contrario, es decir hacer el circuito más pequeño para que quepa en la FPGA pero haciendo que sea más lento.

Estos casos se representan en las gráficas de la figura 8.10. La gráfica A de la izquierda representaría las funciones que muestran la relación entre la superficie y el tiempo de cómputo para dos algoritmos A_0 y A_1. El algoritmo A_0 es menos eficiente que el algoritmo A_1, ya que la implementación del algoritmo A_1 da como resultado un circuito con menos área y menor tiempo de cómputo.

La gráfica B de la derecha muestra las variaciones de área y tiempo de cómputo dentro de un mismo algoritmo. Como muestra la gráfica, empleando un mismo algoritmo podemos movernos dentro de dicha función según dónde tengamos la restricción, en tiempo o en área.

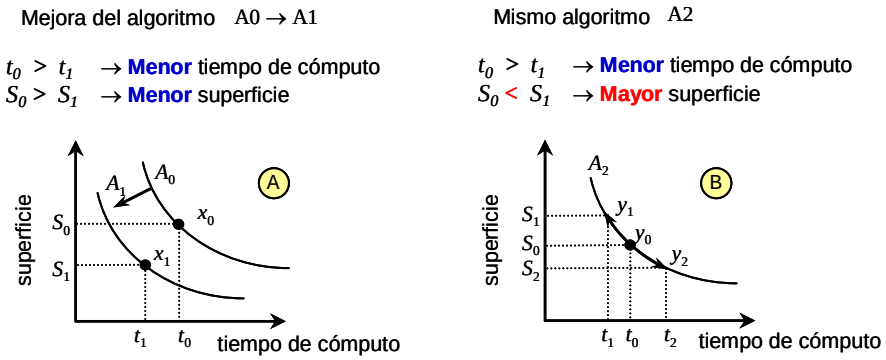


Figura 8.10: Curvas de prestaciones área-tiempo para algoritmos

Estas funciones son simplificaciones para intentar explicar las relaciones entre las prestaciones de un circuito. Evidentemente hay más factores que intervienen.

Para entender esto mejor, resumiremos algunas de las técnicas utilizadas en el diseño de circuitos digitales para aumentar algunas prestaciones a costa de otras. Estas son:

- Paralelización
- Segmentación
- Compartir recursos
- Diseño secuencial

A continuación veremos cada una de estas técnicas con más detalle.

8.9.1.1. Paralelización

Esta técnica consiste en realizar operaciones en paralelo y es una de las grandes ventajas de las FPGAs frente a los microprocesadores. Si queremos aumentar el número de elementos procesados por unidad de tiempo, podemos realizar procesamientos en paralelo, duplicando/triplicando/... los módulos de procesamiento. La figura 8.11 muestra la representación esquemática de esta técnica. Como se puede intuir, el doble de área produce el doble de elementos procesados, que se podría traducir en doble velocidad de procesamiento o mitad de tiempo de cómputo. Sin embargo realmente no es así, pues suele ser más del doble de área al ser necesario incluir lógica de control. Y por otro lado, no es del todo exacto decir que el tiempo de procesamiento se disminuye a la mitad, ya que no siempre habrá dos elementos disponibles para ser procesados.

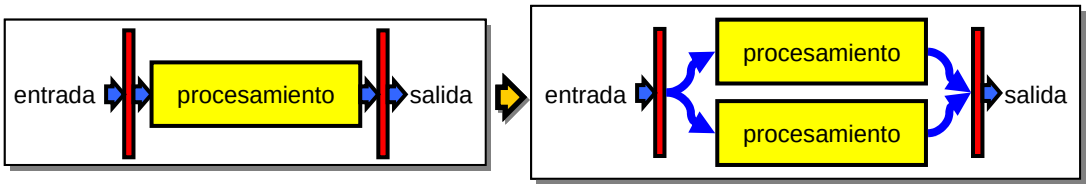


Figura 8.11: Paralelización

8.9.1.2. Segmentación

La segmentación[62] consiste en dividir un procesamiento en tareas más sencillas y separarlas por elementos de memoria. La figura 8.12 muestra la representación esquemática de esta técnica.

[62] La segmentación en inglés se llama *pipelining*

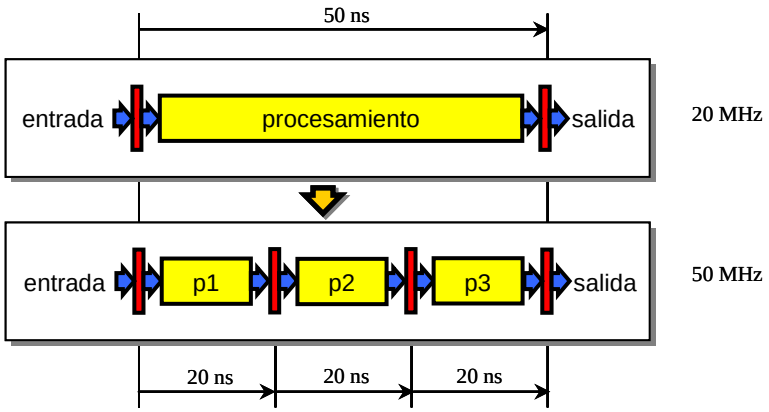


Figura 8.12: Segmentación

Con la segmentación se pueden conseguir dos objetivos:

- Aumentar la frecuencia de reloj del circuito
- Aumentar el cantidad de elementos procesados por unidad de tiempo

Si nos fijamos en la figura 8.12 podemos ver que debido a lo largo que es el procesamiento sin segmentar, la frecuencia máxima del circuito está limitada al tiempo de procesamiento de este módulo. Podría suceder que el resto del circuito tenga que ir más lentamente debido a la lentitud de este módulo, que impone 20 MHz de frecuencia máxima[63]. Esto implicaría un detrimento en las prestaciones del resto del circuito. En cambio, si segmentamos el procesamiento, cada una de las divisiones podrá ir más rápidamente, con lo que la frecuencia máxima del circuito puede ser más elevada (50 MHz en el ejemplo).

Por otro lado, la segmentación permite aumentar la cantidad de elementos procesados ya que el procesamiento se realiza en cadena gracias a la separación que establecen los elementos de memoria. Es decir, al mismo tiempo que *p3* (de la figura 8.12) está terminando de procesar un elemento, *p2* esta procesando el siguiente elemento, y *p1* está procesando el elemento que acaba de entrar en la cadena. Esto implica que, una vez que se ha llenado la cadena, tendremos un nuevo elemento procesado cada 20ns (50 MHz).

Sin embargo la segmentación no es ideal, es decir, si dividimos en tres procesamientos, no obtendremos el triple de frecuencia, ya que es difícil conseguir procesamientos iguales de modo que cada uno de ellos tarde exactamente un tercio del original. Por otro lado, hay que tener en cuenta el llenado y vaciado de la cadena de segmentación, y que cuando tenemos que procesar un sólo elemento va a tardar tres ciclos de reloj.

La segmentación implica un aumento de área por los elementos de memoria añadidos, además suele necesitar alguna lógica añadida de control.

Un ejemplo de multiplicador segmentado se verá en el apartado 8.9.3.

8.9.1.3. Compartir recursos

Cuando el problema es el área y no la velocidad de cómputo, o cuando sabemos que dos módulos del circuito no van a necesitar hacer el mismo procesamiento simultáneamente, se pueden compartir recursos. Por ejemplo, si tenemos dos módulos que necesitan realizar una multiplicación, en vez de que cada uno tenga un multiplicador, podemos diseñar el circuito de modo que haya un único multiplicador para los dos. Cuando es un recurso compartido por muchos módulos, el ahorro de área puede ser importante, aunque

[63] En el caso que compartan el mismo reloj

también puede aumentar los tiempos de procesamiento. También será necesario incluir módulos de control de los recursos, lo que hará que aumente el área y la complejidad del circuito. Esta técnica se emplea extensamente en arquitectura de computadores, en donde se comparten buses, periféricos, unidades de procesamiento, memorias,...

En la figura 8.13 se muestra un esquema del mecanismo de compartir recursos. En el nuevo diseño con el procesamiento compartido se ha añadido un bloque *ctrl* que indica la necesidad de añadir un módulo de control para el recurso compartido y la comunicación con el resto del circuito.

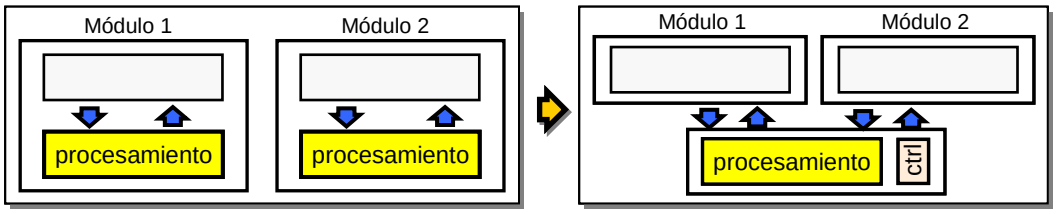


Figura 8.13: Compartir recursos

8.9.1.4. Diseño secuencial

Cuando tenemos que realizar varios procesamientos similares en cadena podemos optar por realizar un diseño secuencial. Con esta alternativa se ahorra área con el coste de tardar varios ciclos de reloj en realizar el procesamiento. En la figura 8.14 se muestra de manera esquemática en qué consiste esta técnica. En esta figura se comparan el diseño combinacional, que es el diseño sin considerar ninguna técnica, con la segmentación y el diseño secuencial. En la figura podemos ver que el mismo procesamiento *P* se repite en varias etapas. Con el diseño segmentado se introducen elementos de memoria entre cada procesamiento, mientras que con el diseño secuencial, se utiliza un sólo módulo de procesamiento *P* que se utiliza de manera iterativa para el procesamiento. Dicho de manera coloquial, el diseño segmentado sería un diseño secuencial desenrollado. Obviamente, el diseño secuencial ahorra módulos de procesamiento pero requiere de un módulo de control de cierta complejidad.

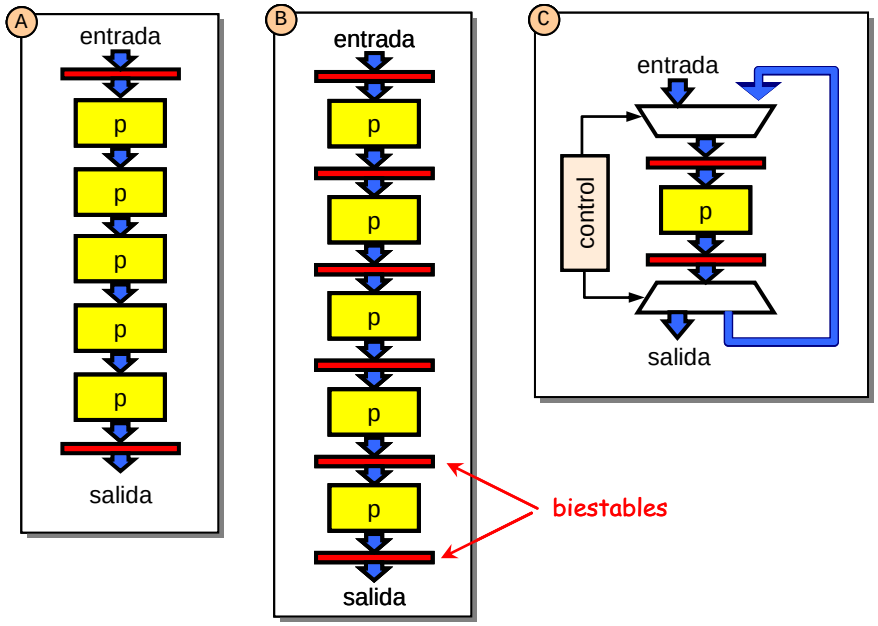


Figura 8.14: Tres opciones de diseño. A: combinacional. B: Segmentado. C: Secuencial

Si observas el módulo combinacional (A) de la figura 8.14 podrás ver que la estructura es similar al multiplicador combinacional de la figura 8.7, siendo el módulo de procesamiento el mostrado en la figura 8.9. En los siguientes apartados se analizarán cada una de estas estrategias con el ejemplo del multiplicador, así aprenderemos a utilizarlas de manera práctica, viendo las ventajas e inconvenientes de cada una.

8.9.2. Análisis del multiplicador combinacional

Los multiplicadores combinacionales suponen un coste considerable en lógica (tamaño) y en retardos para el circuito. Además el aumento del número de bits del multiplicador tiene importantes repercusiones en su tamaño y retardos.

Puede ocurrir que tengamos un circuito descrito correctamente pero que por su gran tamaño no quepa en la FPGA, o también puede ocurrir que los retardos del circuito hagan que no funcione a la frecuencia de nuestra tarjeta. Si esto último ocurriese, el circuito se simularía bien pero no funcionaría correctamente en la FPGA debido a que las señales no llegan a tiempo por sus largos tiempos de propagación. Esto puede ser un gran problema porque podemos estar eternamente revisando la funcionalidad del circuito y la simulación para encontrar los fallos y no lo encontraremos pues funcionalmente el circuito está bien.

En este apartado vamos a analizar el tamaño y los retardos del multiplicador combinacional. Este análisis no sólo es valido para el diseño de multiplicadores sino que será útil para otro tipo de circuitos.

Primero vamos a comparar el área y el camino crítico. Recuerda que el camino crítico es el camino combinacional del circuito que tiene un mayor retardo. El camino crítico será el que determine la frecuencia máxima a la que puede funcionar el circuito.

Para analizar el área de un circuito tomaremos como referencia el número de *slices* que se necesitan. Los *slices* son unas unidades elementales de las FPGAs de Xilinx [26spart3] y contienen los elementos básicos para implementar la lógica circuitos: LUTs, elementos de memoria, multiplexores y lógica aritmética. Cuatro *slices* se agrupan en un bloque lógico configurable o CLB[64].

Gracias a que hemos diseñado el multiplicador combinacional genérico en el apartado 8.8.3 podremos sintetizar el multiplicador cambiando el número de bits de los factores con sólo modificar el genérico `g_bits`.

En el informe de síntesis (recuerda el apartado 8.5.1) puedes comprobar el número de *slices* necesarios para la síntesis y una estimación del retardo máximo. Estos retardos son estimaciones, para obtener un dato más preciso habría que hacer la implementación y el mapeado, e incluso realizar una simulación después de la síntesis. Pero para el análisis que queremos hacer esta estimación es suficiente. Por otro lado, en el caso de que los retardos sean críticos, para reducirlos se pueden incluir restricciones temporales y opciones de síntesis que se enfoquen en minimizarlos.

En la tabla 8.2 se muestra el número de *slices* y los retardos para el multiplicador combinacional según el número de bits. Podemos ver que un único multiplicador de 32 bits ocupa el 11% de la FPGA de la Nexys2 y que los retardos hacen que la frecuencia máxima sea de tan sólo 11,3 MHz.

[64] CLB: *Configurable Logic Block*

nº bits factores	4	8	12	16	20	24	28	32
slices	17	29	67	121	202	290	394	514
% *slices* Nexys2	0,4%	0,6%	1,4%	2,6%	4,3%	6,2%	8,5%	11%
retardo máx. (ns)	15,3	23,9	34,6	45,1	56,9	67,4	77,9	88,4
frecuencia máx. (MHz)	65,3	41,9	28,9	22,2	17,6	14,8	12,8	11,3

Tabla 8.2: Variación de área y retardos con el número de bits de los factores del multiplicador combinacional del apartado 8.8.3 para la Nexys2

Los resultados son muy pobres e implementar este tipo de multiplicadores supone un alto coste en área y una reducción muy importante en la frecuencia. Y lo que es peor, tendríamos que modificar el reloj de la Nexys2 para adaptarnos a esta frecuencia[65].

Los resultados para la XUPV2P tampoco son buenos aunque mejores a causa de sus mayores prestaciones. Como su tamaño es mayor, el multiplicador de 32 bits ocupa un 3,8% de la FPGA y los retardos obligarían a funcionar a la FPGA a 16,3 MHz.

Por suerte, estas FPGAs tienen multiplicadores embebidos de 18 bits. Y son los que se emplean al describir un multiplicador con la operación de multiplicación (código 8.19). Los resultados de área y retardos usando estos multiplicadores son mucho mejores (tabla 8.3). Con estos resultados, para 32 bits la frecuencia máxima (56 MHz) es mayor que la frecuencia del reloj de la Nexys2, por lo que no tendríamos problema al usarlos.

nº bits factores	4	8	12	16	20	24	28	32
slices	0	0	0	0	25	35	45	55
% *slices* Nexys2	0,0%	0,0%	0,0%	0,0%	0,5%	0,8%	1,0%	1,2%
multiplicadores	1	1	1	1	4	4	4	4
% multiplicadores Nexys2	5%	5%	5%	5%	20%	20%	20%	20%
retardo máx. (ns)	9,7	10,2	10,2	10,2	16,4	16,9	17,4	17,9
frecuencia máx. (MHz)	103,4	98,4	98,0	98,0	60,8	59,1	57,5	56,0

Tabla 8.3: Variación de área y retardos con el número de bits de los factores de los multiplicadores embebidos de la Nexys2 (código 8.19)

De la tabla 8.3 podemos observar varias cosas interesantes. A partir de 20 bits se aumenta el número de multiplicadores de uno a cuatro. Esto se debe a que los multiplicadores embebidos son de 18x18 bits y por lo tanto hay usar cuatro para crear un multiplicador mayor. Además, a causa de la lógica que hay que añadir entre los multiplicadores los retardos también aumentan considerablemente. En la figura 8.15 se muestra de manera esquemática cómo se utilizan cuatro multiplicadores de 2x2 para realizar una multiplicación de 4x4. Observa que después de las multiplicaciones hay que realizar la suma de los cuatro resultados. El esquema de la figura es ampliable a otro número de bits.

[65] Recuerda que la frecuencia de reloj de la Nexys2 es de 50 MHz

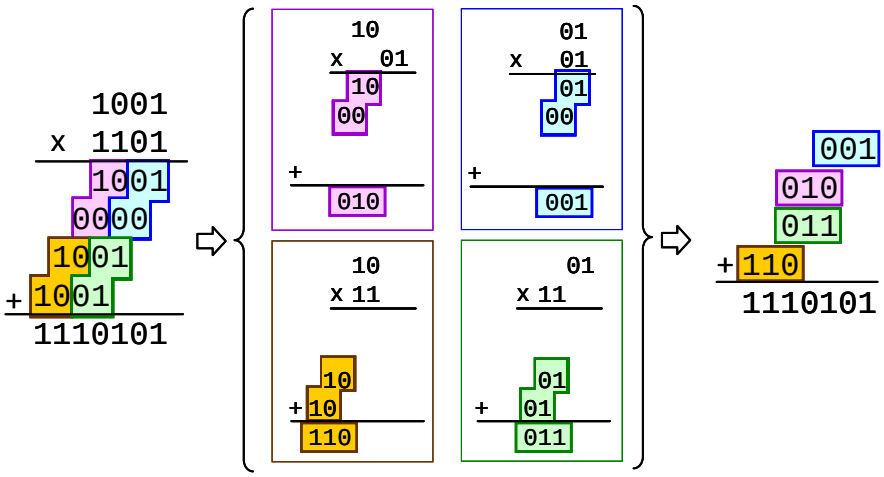


Figura 8.15: Algortimo para calcular un multiplicador de 4x4 a partir de4 multiplicadores de 2x2

Por último, vemos que la *Spartan3e* de la *Nexys2* sólo tiene 20 multiplicadores, por lo que un multiplicador de 20 bits emplea el 20% de los multiplicadores disponibles. La *Virtex2p* de la XUPV2P tiene 138 multiplicadores, por lo que el problema es menor, mientras que la *Spartan3e* de la *Basys* tan sólo tiene 4 multiplicadores.

En el siguiente apartado veremos cómo diseñar un multiplicador segmentado y analizaremos sus prestaciones.

8.9.3. Multiplicador segmentado

En el apartado 8.9.1.2 vimos las ventajas de realizar una segmentación y que esta técnica se podía aplicar a los multiplicadores. Para segmentar un multiplicador tenemos que poner registros entre cada una de las etapas, teniendo cuidado de ir trasladando correctamente los datos de cada etapa.

Teniendo presente el esquema del multiplicador combinacional de la figura 8.7 vemos que se puede dividir naturalmente en tantas etapas como bits de los factores. En la figura 8.7 la línea discontinua azul marca dónde podemos realizar la segmentación.

En la figura 8.16 se muestra el esquema del multiplicador segmentado. Observa que en cada etapa hay registros que separan una etapa de la otra. Incluso los factores (`fact1` y `fact2`) tienen que estar registrados si queremos poder realizar operaciones encadenadas, de manera que en las distintas etapas se puedan estar realizando multiplicaciones simultáneas. Con esto hacemos que cada una de las etapas sea independiente y podamos estar realizando una parte de una multiplicación diferente en cada etapa.

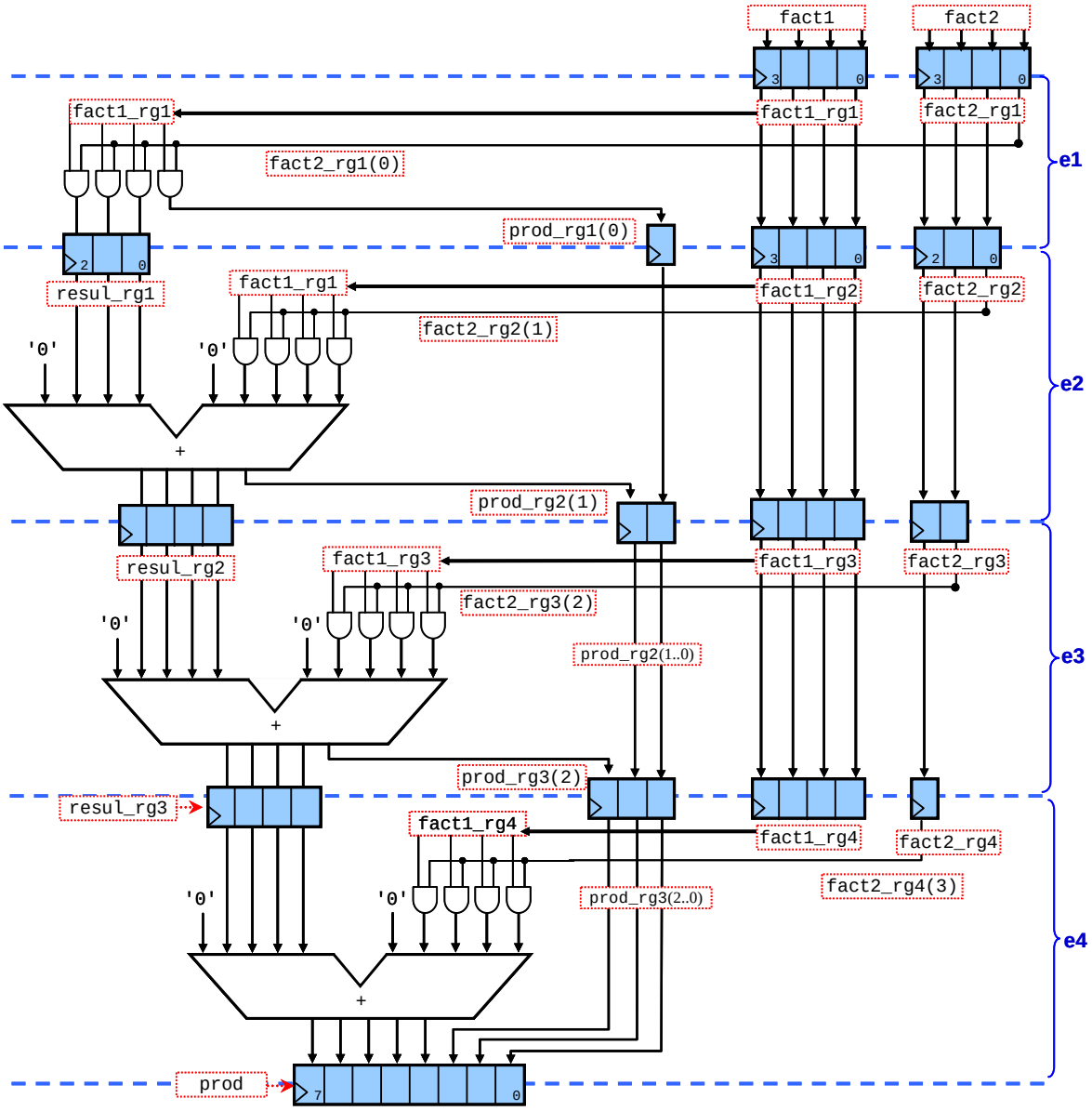


Figura 8.16: Esquema del multiplicador segmentado

En este ejemplo, la multiplicación estará lista en cinco ciclos de reloj, un ciclo más que los bits del multiplicador. Sin embargo, una vez llena la cadena de la multiplicación, si la mantenemos llena obtendremos un resultado distinto cada ciclo de reloj.

Este diseño es modificable, por ejemplo se podría haber eliminado la primera etapa juntándola con la segunda, ya que tiene menor retardo y área.

El diseño VHDL se muestra en el código 8.29. Este diseño se ha hecho genérico y debido a ello hay elementos de señales vectoriales que no se usan, como por ejemplo algunos de los elementos del vector `prod_rg`. Como vemos en la figura 8.16, la primera etapa sólo tiene un elemento: `prod_rg1(0)`, y va aumentando en un elemento con cada etapa. Es decir, en la segunda etapa es `prod_rg2(1 downto 0)`, y así sucesivamente. El diseño VHDL, como es un diseño genérico, se ha creado el vector de vectores `prod_rg`, donde el primer índice corresponde con la primera etapa: `prod_rg(1)(g_bits-1 downto 0)`, pero de este vector sólo se utiliza el bit cero: `prod_rg(1)(0)`. En la segunda etapa, sólo se utilizan los dos bits menos significativos: `prod_rg(2)(1 downto 0)`, y así sucesivamente.

De manera similar ocurre con los registros del segundo factor: `fact2_rg`. Conforme se avanza en las etapas, no es necesario guardar el valor de todos sus elementos. De la figura 8.16 podemos observar que conforme se avanza en las etapas `fact2_rg` disminuye y `prod_rg` aumenta. Así que se podría haber utilizado el mismo vector para los dos, utilizando los elementos inferiores para el `prod_rg` y los superiores para `fact2_rg`. Esto no se ha hecho para no dificultar la comprensión del código. Como consecuencia, al sintetizar el código 8.16 aparecerán algunos avisos de señales no utilizadas.

Para reducir el tamaño del código en este libro no se ha incluido el reset en los biestables del multiplicador, pero sería conveniente incluirlo.

```
entity mult_seg_gen is
 Generic ( g_bits   : natural := 4 );
 Port (
   clk     : in  std_logic;
   fact1   : in  unsigned (g_bits-1 downto 0);
   fact2   : in  unsigned (g_bits-1 downto 0);
   prod    : out unsigned (2*g_bits-1 downto 0)
  );
end mult_seg_gen;

architecture behavioral of mult_seg_gen is
  type t_vector_fact  is array (1 to g_bits) of unsigned (g_bits-1 downto 0);
  type t_vector_resul is array (1 to g_bits) of unsigned (g_bits downto 0);

  -- prod_rg es la mitad inferior de prod, que se va manteniendo para abajo
  signal prod_rg, sum_anded, fact1_rg, fact2_rg, resul_rg  : t_vector_fact;
  signal resul        : t_vector_resul;
begin
-- primera etapa ---------------------------------------------------------
  sum_anded(1) <= fact1_rg(1) when fact2_rg(1)(0) = '1' else (others => '0');
  resul(1)     <= '0' & sum_anded(1);

  P_reg_1e: Process (clk)
  begin
    if clk'event and clk='1' then
      fact1_rg(1)   <= fact1;  -- registramos entradas
      fact2_rg(1)   <= fact2;
      resul_rg(1)   <= resul(1)(g_bits downto 1);  -- registramos resultados
      prod_rg(1)(0) <= resul(1)(0);
    end if;
  end process;

-- resto de etapas -------------------------------------------------------
  gen_mult: for i in 2 to g_bits generate
    sum_anded(i) <= fact1_rg(i) when fact2_rg(i)(i-1)='1' else (others =>'0');
    resul(i) <= ('0' & resul_rg(i-1)) + ('0' & sum_anded(i));

    P_resuli_rg: Process (clk)
    begin
      if clk'event and clk='1' then
        fact1_rg(i)     <= fact1_rg(i-1);
        fact2_rg(i)     <= fact2_rg(i-1);
        resul_rg(i)     <= resul(i)(g_bits downto 1);
        prod_rg(i)(i-1) <= resul(i)(0);
        prod_rg(i)(i-2 downto 0) <= prod_rg(i-1)(i-2 downto 0);
      end if;
    end process;
  end generate;

-- registrar la salida ---------------------------------------------------
  P_prod_rg: Process (clk)
  begin
    if clk'event and clk='1' then
      prod(2*g_bits-1 downto g_bits) <= resul(g_bits)(g_bits downto 1);
    end if;
  end process;
  -- prod_rg ya esta registrado
  prod(g_bits-1 downto 0) <= prod_rg(g_bits)(g_bits-1 downto 0);
end behavioral;
```

Código 8.29: Multiplicador segmentado genérico

De la misma manera que en los multiplicadores anteriores, se ha analizado el tamaño y los retardos del multiplicador segmentado (tabla 8.4). Como era de esperar el área aumenta respecto al multiplicador combinacional (tabla 8.2) y la frecuencia ha disminuido enormemente. Por tanto, hemos resuelto el problema de la frecuencia de reloj, pero a costa de aumentar el área.

nº bits factores	4	8	12	16	20	24	28	32
slices	30	83	178	306	492	698	953	1248
% *slices* Nexys2	0,6%	1,8%	3,8%	6,6%	10,6%	15,8%	20,5%	26,8%
retardo máx. (ns)	4,7	4,1	4,5	4,8	4,7	5,0	5,1	5,4
frecuencia máx. (MHz)	214,3	246,2	222,2	207,9	211,2	198,5	195,4	185,6

Tabla 8.4: Variación de área y retardos con el número de bits de los factores de los multiplicadores segmentados (código 8.29)

Como los retardos han disminuido mucho más de lo que necesitamos, y para no aumentar tanto el área utilizada, en vez de realizar la segmentación entre cada etapa podemos realizarla cada cierto número de etapas. Para este caso parece que segmentar cada tres o cuatro etapas mantendría la frecuencia superior a 50 MHz y haría disminuir las necesidades de área.

La segmentación también es útil cuando queremos implementar multiplicadores mayores de 18 bits utilizando los multiplicadores embebidos. Como vimos en la tabla 8.3, los retardos aumentaban por la necesidad de unir los resultados de los cuatro multiplicadores (figura 8.15). Segmentar entre la salida de los multiplicadores y las sumas posteriores haría disminuir los retardos, consiguiéndose frecuencias de reloj más altas.

8.9.4. Implementación del multiplicador secuencial

Se podría decir que el multiplicador secuencial es como el multiplicador segmentado enrollado sobre sí mismo, de modo que las salidas de cada etapa se vuelven a sumar en el único sumador en vez de ir a la siguiente etapa (recuerda la figura 8.14). Por lo tanto, todas las etapas se procesan en el mismo hardware y como consecuencia el multiplicador estará ocupado hasta que termine.

Será necesario un control que cuente el número de la multiplicación parcial en la que se está, que indique cuándo ha terminado y que controle cuándo se pueden admitir datos para una nueva multiplicación. La figura 8.17 muestra un esquemático de uno de los posibles circuitos que implementa un multiplicador secuencial. No es un esquemático completamente detallado para que sea más entendible.

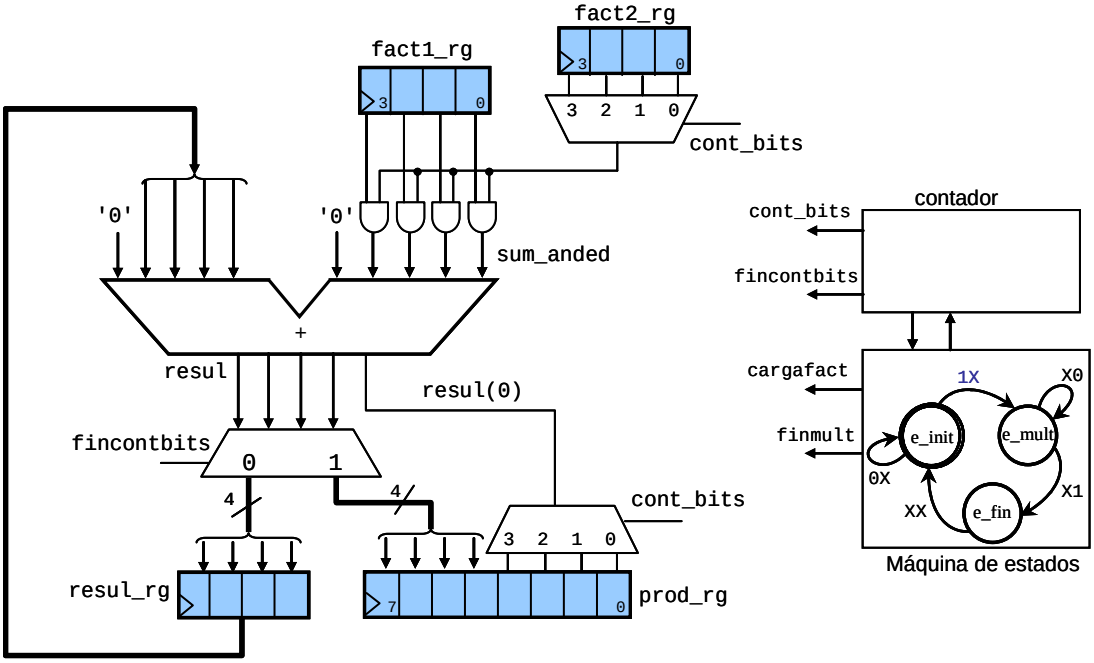


Figura 8.17: Esquemático del multiplicador secuencial de 4 bits

En la figura 8.17 se puede identificar el elemento básico de procesamiento sobre el que se iteran las operaciones y que ya vimos en la figura 8.9.

La descripción VHDL correspondiente al multiplicador secuencial se muestra en el código 8.30.

```
entity mult_seq_gen is
 Generic (g_bits   : natural := 4 );
 Port (
   rst     : in  std_logic;
   clk     : in  std_logic;
   start   : in  std_logic;  -- orden de empezar multiplicacion
   fact1   : in  unsigned (g_bits-1 downto 0);
   fact2   : in  unsigned (g_bits-1 downto 0);
   finmult : out std_logic;  -- fin de multiplicacion
   prod    : out unsigned (2*g_bits-1 downto 0)
  );
end mult_seq_gen;

architecture behavioral of mult_seq_gen is
  constant c_nb_contbits : natural := log2i(g_bits-1);
  signal   contbits_us   : unsigned (c_nb_contbits downto 0);
  signal   contbits      : natural range 0 to 2**(c_nb_contbits+1)-1;
  signal   fincontbits   : std_logic;
  signal   cargafact     : std_logic;
  signal   fact1_rg, fact2_rg, sum_anded  : unsigned (g_bits-1 downto 0);
  signal   resul         : unsigned (g_bits downto 0);
  -- resul_rg tiene 1 bit menos, porque no guarda el bit menos significativo
  -- ya que este va a prod_rg
  signal   resul_rg      : unsigned (g_bits-1 downto 0);
  signal   prod_rg       : unsigned (2*g_bits-1 downto 0);
  type     estados_mult is (e_init, e_mult, e_fin);
  signal   estado_act, estado_sig  : estados_mult;
begin
```

```
-------------------- MAQUINA DE ESTADOS ----------------------------------
  P_control_seq: Process(rst,clk)
  begin
    if rst = c_on then
      estado_act <= e_init;
    elsif clk'event and clk='1' then
      estado_act <= estado_sig;
    end if;
  end process;

  P_control_comb: Process (estado_act, start, fincontbits)
  begin
    cargafact <= '0';
    finmult    <= '0';
    case estado_act is
      when e_init =>
        if start = '1' then
          estado_sig <= e_mult;
          cargafact  <= '1';
        else
          estado_sig <= e_init;
        end if;
      when e_mult =>
        if fincontbits = '1' then
          estado_sig <= e_fin;
        else
          estado_sig <= e_mult;
        end if;
      when e_fin =>
        estado_sig <= e_init;
        finmult    <= '1';
    end case;
  end process;

--------------- CARGA Y REGISTRO DE LOS FACTORES A MULTIPLICAR-------------------
  P_cargafact: Process (rst, clk)
  begin
    if rst = c_on then
      fact1_rg <= (others => '0');
      fact2_rg <= (others => '0');
    elsif clk'event and clk='1' then
      if cargafact = '1' then
        fact1_rg <= fact1;
        fact2_rg <= fact2;
      elsif estado_act = e_fin then
        fact1_rg <= (others => '0');
        fact2_rg <= (others => '0');
      end if;
    end if;
  end process;

-------------- CUENTA LOS BITS QUE SE VAN MULTIPLICANDO --------------------------
  P_cuenta_bits: Process (rst, clk)
  begin
    if rst = c_on then
      contbits_us <= (others =>'0');
    elsif clk'event and clk='1' then
      if estado_act = e_mult then
        contbits_us <= contbits_us + 1;
      else
        contbits_us <= (others =>'0');
      end if;
    end if;
  end process;

  fincontbits <= '1' when contbits=g_bits-1 else '0';

  contbits <= to_integer(contbits_us); -- para los rangos se necesita un entero

--------------- MODULO DE LA SUMA CON MULTIPLICACION (figura 8.9)
  -- la multiplicacion parcial del bit de fact2 con fact1,
  -- teniendo en cuenta el desplzamiento
  sum_anded <= fact1_rg when fact2_rg(contbits) = '1' else (others => '0');
  resul     <= ('0' & resul_rg) + ('0' & sum_anded);
```

```
--------------------- GUARDA EL RESULTADO en:
-- resul_rg: si no se ha terminado
-- prod_rg : si es el resultado definitivo (fincontbits='1')
  P_acum_mult: Process (rst, clk)
  begin
    if rst = c_on then
      resul_rg <= (others => '0');
      prod_rg  <= (others => '0');
    elsif clk'event and clk='1' then
      case estado_act is
        when e_init | e_fin =>
          resul_rg <= (others => '0');
          prod_rg  <= (others => '0');
        when e_mult =>
          if fincontbits = '0' then
            resul_rg <= resul(g_bits downto 1);
            prod_rg(contbits) <= resul(0);
          else
            resul_rg <= (others => '0'); -- ya no hace falta resul_rg
            prod_rg(2*g_bits-1 downto contbits) <= resul; --se guarda en prod_rg
          end if;
      end case;
    end if;
  end process;

  prod <= prod_rg;
end behavioral;
```

Código 8.30: Multiplicador secuencial genérico

Por último nos queda analizar las prestaciones de este multiplicador. La tabla 8.5 muestra que con este multiplicador mantenemos el tamaño controlado y también que la frecuencia no baja excesivamente. Sin embargo hay que tener en cuenta que la multiplicación tarda un número de ciclos mayor que el número de bits del multiplicador[66].

nº bits factores	4	8	12	16	20	24	28	32
slices	38	37	83	81	136	144	156	162
% *slices* Nexys2	0,8%	0,8%	1,8%	1,7%	2,9%	3,1%	3,4%	3,5%
retardo máx. (ns)	6,5	7,9	9,1	8,9	11,4	11,3	11,3	10,8
frecuencia máx. (MHz)	152,7	126,5	110,3	112,4	88,0	88,1	88,2	92,4

Tabla 8.5: Variación de área y retardos con el número de bits de los factores de los multiplicadores secuenciales (código 8.30)

8.9.5. Comparación de los multiplicadores

Finalmente compararemos las cuatro alternativas de multiplicadores implementados. Recordemos antes cuáles son estas alternativas:

- Multiplicador embebido (código 8.19)
- Multiplicador combinacional (figura 8.7 y código 8.22). El multiplicador genérico combinacional (apartado 8.8.3) y el estructural (apartado 8.8.4) también están dentro de esta categoría
- Multiplicador segmentado (apartado 8.9.3)
- Multiplicador (apartado 8.9.4)

En la figura 8.18 se muestra de manera gráfica el porcentaje de área (*slices*) utilizado en la Nexys2 para implementar los distintos multiplicadores según el número de bits de los factores.

[66] Dependiendo del tipo de control y el tipo de máquina de estados el número de ciclos puede puede variar.

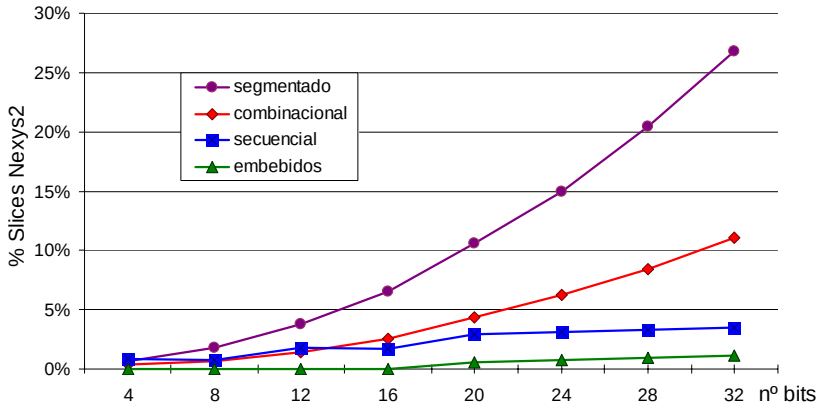


Figura 8.18: Utilización de área de la Nexys2 para implementar los distintos multiplicadores respecto al número de bits del multiplicador

En la gráfica se puede ver que la mejor alternativa en cuanto área es utilizar los multiplicadores embebidos (para eso están). El inconveniente es que la *Spartan3e* de la Nexys2 tiene sólo 20 multiplicadores, y cuando se realiza una multiplicación de 20 bits, se utilizan 4 de ellos (el 20%). Con lo que sólo podríamos implementar cinco multiplicadores embebidos de 20 bits.

Los multiplicadores secuenciales mantienen el área controlada, mientras que los multiplicadores combinacionales y segmentados aumentan mucho su área conforme aumenta el número de bits, especialmente los segmentados.

En la figura 8.19 se muestra la frecuencia máxima a la que puede funcionar la Nexys2 según el multiplicador que se use y el número de bits de los factores.

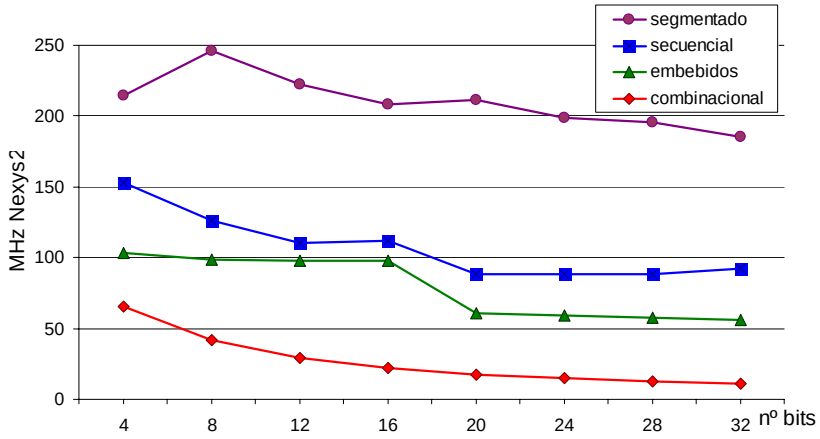


Figura 8.19: Frecuencia máxima de la Nexys2 para los distintos multiplicadores respecto al número de bits del multiplicador

De la gráfica podemos extraer que los multiplicadores combinacionales tienen muchos retardos (baja frecuencia) mientras que en el extremo opuesto, los segmentados tienen una frecuencia mucho mayor que la que normalmente utiliza la tarjeta (50MHz). Por lo tanto, como ya se ha indicado, para aumentar la frecuencia de los combinacionales y reducir el área de los segmentados, se puede hacer una segmentación intermedia.

El multiplicador secuencial tiene buenas características en cuanto a retardos y área, pero es lento porque requiere varios ciclos de reloj.

Y por último, el multiplicador embebido tiene buenas prestaciones, pero podemos observar que la frecuencia baja a partir de 18 bits, ya que hay que utilizar varios

multiplicadores (figura 8.15). Para solucionar esto se puede segmentar a la salida de los multiplicadores embebidos o incluso realizar diseños mixtos, usando multiplicadores embebidos a la vez que multiplicadores combinacionales o segmentados.

Como resumen, no existe una estrategia que sea la mejor en todos los casos, es la tarea del diseñador analizar los requisitos del sistema y elegir lo que más conviene en cada caso. También es importante considerar el tipo de algoritmo, ya que éste puede influir en las prestaciones del sistema independientemente de la estrategia utilizada. Hay otros algoritmos y variantes que mejoran los que hemos vistos, el estudio de éstos se sale del objetivo de este libro, para más información puedes consultar la referencia [7desch].

Por otro lado también hay que señalar que el tiempo de diseño es un factor muy importante. Esto significa que no hay que mejorar las prestaciones cuando no es necesario. Por ejemplo, si nuestro multiplicador funciona bien y da las prestaciones requeridas, no hay que dedicar más tiempo a mejorarlo. No es eficiente implementar un multiplicador más rápido o que ocupe menos área si para su diseño necesito emplear el doble de tiempo y mi sistema final no obtiene ninguna ventaja. Un retraso en el proyecto puede ser peor que reducir en un 5% el área utilizada si tenemos una FPGA suficientemente grande. Es la tarea del ingeniero valorar todos los aspectos del diseño, sus costes y los tiempos de desarrollo.

8.10. División

Al contrario de la multiplicación, si necesitamos implementar una división entera, el operador "/" no está soportado para la síntesis, por lo tanto tendremos que diseñar el divisor[67].

Existen muchos algoritmos de división, a continuación se explicará uno que es el que resulta más parecido a nuestra forma de dividir a mano y es por tanto más sencillo de entender.

En la figura 8.20 se muestra una división entera realizada con números enteros decimales y con números enteros binarios.

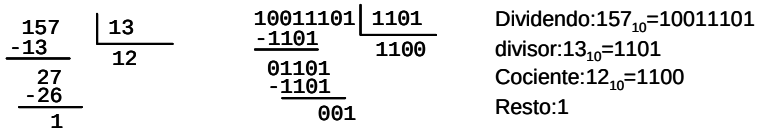


Figura 8.20: Ejemplo de división entera en decimal y en binario

Veamos paso a paso el algoritmo para hacer la división:

Primero desplazamos a la izquierda el divisor lo más que podamos siempre que "quepa" en el dividendo (D>d). La cantidad de veces que se ha desplazado la memorizamos. En electrónica digital, diríamos que la guardamos en un registro que llamaremos `Desplz`. En nuestro ejemplo de la figura 8.21 el desplazamiento es 3.

[67] A no ser que estemos haciendo una división entre dos constantes. En este caso el compilador VHDL calcula la división y el resultado es el que se usa en la síntesis. O que queramos hacer una división por una potencia de dos.

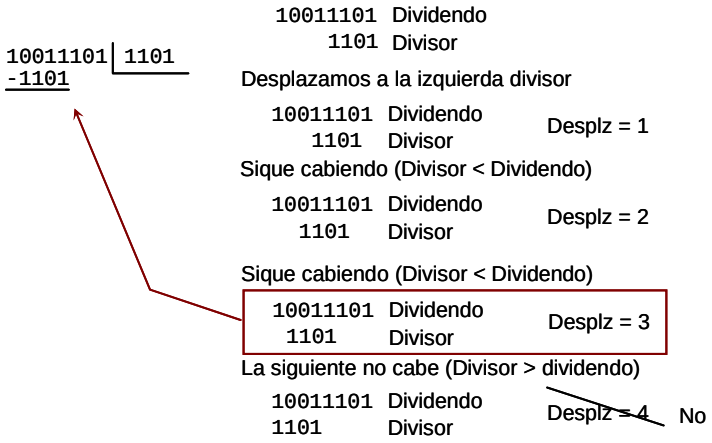


Figura 8.21: Paso 1 de la división: cálculo del desplazamiento inicial

Una vez que tenemos el divisor desplazado lo más posible a la izquierda (`Desplz=3`), restamos al dividendo el divisor y ponemos un 1 en el cociente (figura 8.22). En la figura se han rellenado los ceros que faltan a la derecha provocados por el desplazamiento. En el cálculo manual se hace implícitamente aunque no se ponen para ahorrar la escritura de la resta completa.

```
10011101 | 1101
-1101000 |  1          Desplz = 3
 0110101
```

Figura 8.22: Paso 2 de la división

Ahora, el divisor, que hemos desplazado tres veces a la izquierda, lo desplazamos una vez a la derecha, es decir, como si del original lo desplazásemos dos veces a la izquierda. Por tanto `Desplz=2`. Probamos si cabe con el resultado de la resta anterior, si cabe (D≥d) ponemos un uno en el cociente.

```
10011101 | 1101
-1101000 |  11         Desplz = 2
 0110101
 -110100
  000001
```

Figura 8.23: Paso 3 de la división

Volvemos a desplazar el divisor a la derecha (ahora `Desplz=1`), y volvemos a ver si cabe en la nueva resta que hemos hecho.

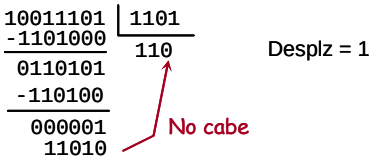


Figura 8.24: Paso 4 de la división

Vemos que esta vez no cabe, por tanto pondremos un cero en el cociente, no hacemos la resta, y probamos desplazando el divisor a la derecha (ahora el `Desplz=0`, volvemos a como estaba al principio)

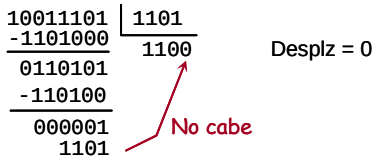


Figura 8.25: Paso 5 de la división

Tampoco cabe, y como el cociente ha vuelto a su posición original (`Desplz = 0`) la división se termina (pues no sacamos decimales).

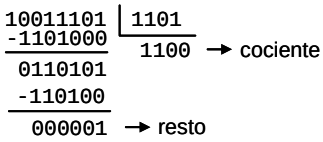


Figura 8.26: Paso 6 de la división

Ahora desplazando los restos a la izquierda podríamos sacar decimales. En este caso podríamos sacar el primer decimal para calcular el redondeo del cociente.

Así pues la implementación en hardware de este algoritmo de división se podría hacer de manera secuencial, pero también se podría "desenrollar" y realizarla de manera combinacional o segmentada.

Si se realiza de manera secuencial, la división se haría en varios ciclos de reloj, que dependerá del número de bits de los operandos. Por tanto, igual que con el multiplicador secuencial, el divisor necesitará de una parte de control que ordene la ejecución de las operaciones que haya que realizar en cada momento. Una vez terminadas las operaciones deberá proporcionar una señal de aviso que indique que el resultado está disponible. Además, el módulo tendrá una señal de entrada que ordene la ejecución de la división.

Para este ejemplo no se va a incluir el esquema o el código VHDL del divisor. Te recomendamos que lo implementes por ti mismo y que hagas el banco de pruebas. Con la información sobre el algoritmo de división junto con las explicaciones del algoritmo de multiplicación y los otros operadores de este capítulo, pensamos que lo podrías hacer por ti mismo.

Incluso te proponemos que implementes una pequeña calculadora en la FPGA. Metiendo los datos por los pulsadores e interruptores o por la UART. Por ejemplo, si usas la *Nexys2*, los interruptores podrían ser los operandos y los botones la operación: suma, resta, multiplicación y división. El resultado se obtendría por el display de siete segmentos.

Si usas la XUPV2P, como no hay tantos pulsadores ni interruptores, puedes realizar la operación por la UART, introduciendo primero el primer operador, luego la operación y por último el segundo operador. El mayor problema aquí será el control de lo que llega por la UART y la conversión en órdenes al módulo que realiza la operación. Para simplificar, te recomendamos que los números que envíes estén codificados en binario o en hexadecimal, pero no en decimal. Esto significa que desde el hiperterminal[68] enviarías por ejemplo: "1110/11=" ó "E/3=" en vez de "14/3=". Tu circuito debería devolver: "1000 R10"(en binario) ó "4 R2" (en hexadecimal). Elige una de las dos formas.

Este circuito de división o calculadora por la UART tiene cierta complejidad debido al control, conversiones y la unión de varios módulos, aparte de la propia complejidad del

[68] Recuerda el apartado 7.6.3

divisor. Sin embargo, podría ser un interesante proyecto que unificaría todo lo que hemos visto hasta ahora.

9. Controlador de pantalla VGA

En esta práctica aprenderemos a diseñar un controlador para pantalla. Este controlador nos será muy útil para las siguientes prácticas, ya que permitirá al sistema aumentar de manera extraordinaria las capacidades de ofrecernos información

Antes de explicar el funcionamiento de un controlador, piensa cómo puede un único circuito indicar el valor de cada uno de los píxeles de una pantalla. Incluso en pantallas de baja resolución como las de 640x480 (640 columnas y 480 filas) tenemos más de trescientos mil píxeles.

Evidentemente no se pueden controlar todos a la vez, y el método par a controlarlo es similar al control del display de siete segmentos que se vio en el capítulo 6 del libro [17mach]. Es decir, multiplexando en el tiempo. Cada píxel se activa durante una pequeña fracción de tiempo y se refresca continuamente. Como esto ocurre a una velocidad muy elevada, nosotros no nos damos cuenta y creemos que están continuamente encendidos.

Para controlar una la pantalla existen multitud de estándares. Mediante SVGA (*Super Video Graphics Array*) se conocen múltiples de estándares de visualización gráfica de ordenadores. SVGA surgió después del estándar VGA, y muchas veces se nombran de manera indistinta (igual que otras variantes como XSGA). Estas propuestas no eran únicas y cada fabricante proponía temporizaciones diferentes. Posteriormente, VESA (*Video Electronics Standards Association*: http://www.vesa.org) definió un estándar para el video con el que se intentó aglutinar las propuestas de los fabricantes.

El conector VGA es del tipo mostrado en la figura 9.1, que es habitual en los ordenadores actuales. Las señales y el control se definieron pensando en monitores de rayos catódicos, sin embargo, los monitores digitales se han adaptado al mismo estándar para evitar problemas de compatibilidad.

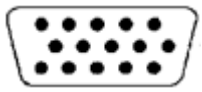


Figura 9.1: Conector VGA

Para transformar las señales digitales de la FPGA en analógicas de estándar VGA se necesitan varios conversores digital-analógicos. Estos conversores suelen estar integrados en un chip y pueden ser bastante diferentes unos de otros. Esto ocurre con los de la tarjeta XUPV2P y la Nexys2.

En este capítulo veremos el estándar VGA. A continuación se resumirán las características de los convertidores digital-analógicos de las placas *XUPV2P* y *Nexys2*. Y por último propondrá el diseño de un controlador VGA.

9.1. Funcionamiento del estándar VGA

Para el control del monitor, la información de cada píxel de la pantalla se va enviando consecutivamente al ir barriendo toda la superficie por líneas horizontales y verticales. Cada píxel se compone de tres colores: rojo, verde y azul.

Las tres primeras señales de la tablas 9.2 y 9.3 indican la intensidad de cada color. Por ejemplo, si `red=0` el píxel no contiene rojo, si por el contrario `red=255` (o `red=7` en la Nexys2) el píxel tiene el máximo nivel de rojo.

El envío de los valores de los píxeles se hace fila a fila empezando por el píxel de arriba a la izquierda y terminando con el píxel de abajo a la derecha (figura 9.2). Existen distintas resoluciones de pantalla, por ejemplo de 640 columnas x 480 filas (640x480) y otras más habituales como 800x600 ó 1024x768.

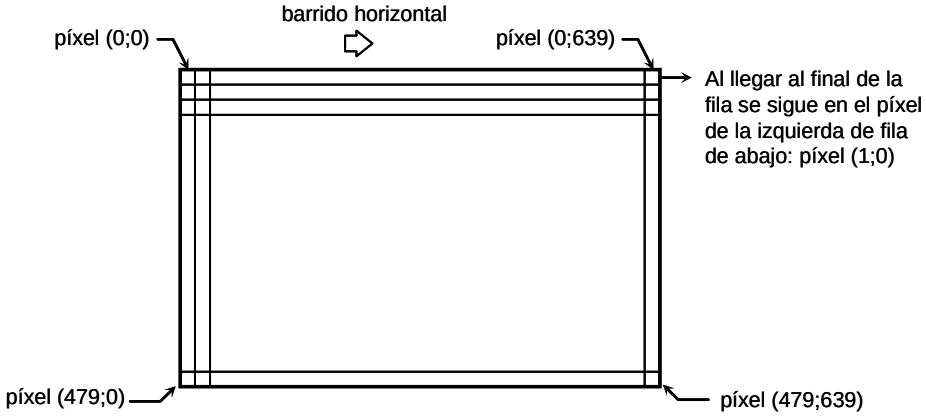


Figura 9.2: Píxeles en una pantalla con resolución de 640x480

Para indicar el cambio de fila y el cambio de pantalla existen dos señales de sincronismo: sincronismo horizontal (hsynch) y vertical (vsynch). El sincronismo horizontal marca el final de una línea y el comienzo de una nueva. El sincronismo vertical marca el fin de una pantalla y el comienzo de una nueva. La frecuencia de refresco viene dada por la frecuencia del sincronismo vertical.

La figura 9.3 muestra las señales de sincronismo para el control de una pantalla VGA con 640x480 píxeles y una frecuencia de refresco de aproximadamente 60 Hz. Los tiempos se han adaptado a una frecuencia de 25 MHz: cada píxel dura 40ns.

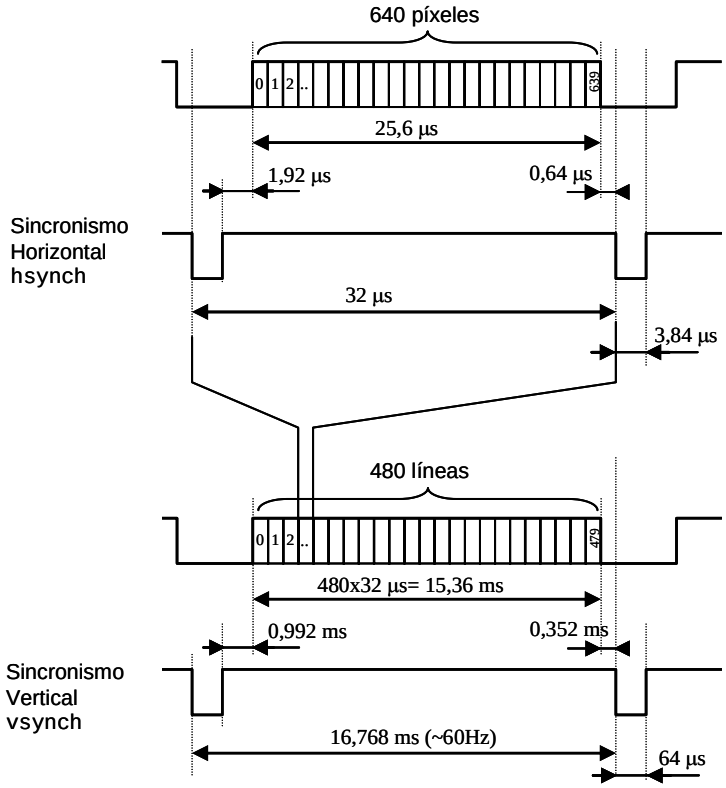


Figura 9.3: Señales de sincronismo para una frecuencia de refresco de 60 Hz y 640x480 píxeles

Los valores de los tiempos son aproximados. Existe un estándar que define diversas frecuencias a las que normalmente se adaptan lo monitores, aunque como ya se ha dicho que varían según el fabricante. Observa que hay intervalos de tiempo entre la señal de sincronismo y el envío de la información de los píxeles, a estos se les llama porches delantero y trasero. En la figura 9.4 se muestran el porche trasero y delantero (*back porch* y *front porch*).

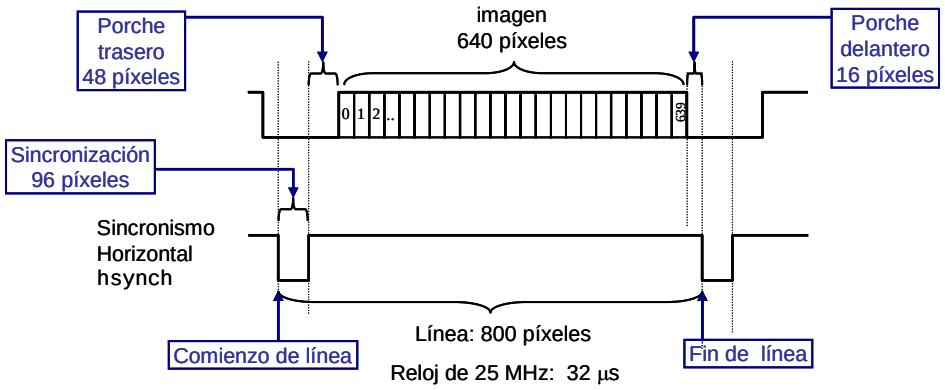


Figura 9.4: Cronograma de una línea en píxeles

A veces, en el diseño digital las especificaciones no se dan en tiempos, sino que se da la frecuencia a la que salen los píxeles. Con esta frecuencia, se especifica el número de píxeles de los porches y las señales de sincronización para la sincronización horizontal, y el número de líneas para la vertical. Por ejemplo, en la figura 9.4 se muestra la especificación de las líneas en píxeles y no en tiempos.

En la tabla 9.1 se muestran los valores para diversas resoluciones de pantalla. Todas las filas excepto la primera se corresponden con formatos propuestos por VESA. La primera fila de la tabla (sombreada) es una aproximación de la segunda fila y es la que usaremos en nuestro diseño, para evitar una frecuencia de reloj de 25,175 MHz. En la figura 9.3 se han obtenido los tiempos tomando los mismos valores de píxeles de la segunda fila, aunque con una frecuencia de reloj ligeramente menor: 25 MHz, en vez de 25,175 MHz. Ya que, de hecho, con un reloj de 25 MHz se pueden tomar los valores de la segunda fila, y el controlador también funcionará en la mayoría de los monitores.

Formato	Reloj MHz	Horizontal (en píxeles)					Vertical (en líneas)				
		Vídeo activo	Porche delantero	Sincr.	Porche trasero	Total	Vídeo activo	Porche delantero	Sincr.	Porche trasero	Total
640x480@60Hz	25	640	16	96	48	800	480	9	2	29	520
640x480@60Hz	25,175	640	16	96	48	800	480	11	2	31	524
800x600@60Hz	40,000	800	40	128	88	1056	600	1	4	23	628
800x600@72Hz	50,000	800	56	120	64	1040	600	37	6	23	666
1024x768@60Hz	65,000	1024	24	136	160	1344	768	3	6	29	806
1024x768@75Hz	75,000	1024	24	136	144	1328	768	3	6	29	806

Tabla 9.1: Valores para diversas resoluciones de pantalla

9.2. Conversores digital-analógico para VGA

Como estamos utilizando dos tarjetas electrónicas, a continuación se resumen los convertidores de estas dos placas y sus entradas.

9.2.1. Conversor VGA de la XUPV2P

La tarjeta *XUPV2P* tiene un triple conversor digital-analógico destinado a aplicaciones de video (*FSM3818*). Las salidas digitales de la FPGA correspondientes a cada color pasan por este circuito integrado y de allí van al conector VGA. En esta tarjeta se utilizan 8 bits para cada color: rojo, verde y azul (RGB[69]). Por tanto, como cada color tiene 8 bits, tendremos 256 niveles distintos para cada color. Contando los tres colores, se tienen más de 16,7 millones de colores (256^3).

Los pines de la FPGA que controlan el video se muestran a continuación. Más adelante se explicará su función con más detalle.

Puerto	bits	I/O	pin	Descripción
`red`	8	O		Nivel de rojo del píxel (de 0 a 255). Pines: H10(bit 7); C7; D7; F10; F9; G9; H9; G8 (bit 0)
`green`	8	O		Nivel de verde del píxel (de 0 a 255). Pines: E11(bit 7); G11; H11; C8; D8; D10; E10; G10
`blue`	8	O		Nivel de azul del píxel (de 0 a 255). Pines: E14 (bit7); D14; D13; C13; J15; H15; E15; D15
`pxl_clk`	1	O	`H12`	Reloj del conversor, que tendrá una frecuencia diferente según la resolución y frecuencia refresco escogidas (tabla 9.1)
`vga_blank`	1	O	`A8`	Si está a cero hace que no se saque nada por pantalla (para cuando no se usa). Si vale uno, funciona normalmente. En la zona en la que no se emite información visible se pone a cero.
`hsynch`	1	O	`B8`	Sincronismo horizontal. Indica el cambio de fila. Según el estándar que se use, esta señal puede funcionar a nivel alto o bajo.
`vsynch`	1	O	`D11`	Sincronismo vertical. Indica el cambio de pantalla (frecuencia de refresco). Según el estándar que se use, esta señal puede funcionar a nivel alto o bajo
`comp_synch`	1	O	`G12`	Con un '1' se añade corriente extra al conversor digital analógico del verde. Si no se activa, los blancos se ven rosados porque el verde no tiene tanta intensidad. Se pone a cero en los sincronismos[70] (horizontales y verticales). Sin embargo, cuando se activa, a veces se ven brillos verdes extraños, así que al final puede salir mejor ponerla siempre a cero.

Tabla 9.2: Puertos del conversor VGA de la XUPV2P

9.2.2. Conversor VGA de la Nexys2

La tarjeta *Nexys2* tiene un conversor digital-analógico más sencillo, con solo 10 bits frente a los 29 bits de la tabla 9.2. La *Nexys2* tiene 8 bits para todos los colores, lo que hace que sólo tenga 256 colores distintos. Se emplean 3 bits para el rojo, 3 bits para el verde y 2 bits para el azul, ya que parece que el ojo humano tiene menos sensibilidad para el azul. Además de estos 8 bits, están las señales de sincronismo vertical y horizontal, pero no se proporciona reloj como en la *XUPV2P* ni otras señales como `vga_blank` y `comp_synch`.

Puerto	bits	I/O	pin	Descripción
`red`	3	O		Nivel de rojo del píxel (de 0 a 7 niveles). Pines: R8 (bit 2, más significativo); T8; R9 (bit 0, menos significativo)

[69] RGB: de las siglas en inglés de los colores: *red, green* y *blue*

[70] Independientemente del nivel de los sincronismos `hsynch` y `vsych`

green	3	O		Nivel de verde del píxel (de 0 a 7 niveles). Pines: P6 (bit 2, más significativo); P8; N8 (bit 0, menos significativo)
blue	2	O		Nivel de azul del píxel (de 0 a 3 niveles). Pines: U4 (bit 1, más significativo); U5 (bit 0, menos significativo)
hsynch	1	O	T4	Sincronismo horizontal. Indica el cambio de fila. Según el estándar que se use, esta señal puede funcionar a nivel alto o bajo.
vsynch	1	O	U3	Sincronismo vertical. Indica el cambio de pantalla (frecuencia de refresco). Según el estándar que se use, esta señal puede funcionar a nivel alto o bajo

Tabla 9.3: Puertos del conversor VGA de la Nexys2

De estas tablas se puede observar que los conversores son bastante distintos.

9.3. Módulo de sincronización

Con la información que tenemos podemos realizar un módulo que genere las señales de sincronización para la pantalla VGA. Haremos un controlador con una resolución de 640x480 píxeles a 25 MHz cada píxel, esto es, utilizaremos las especificaciones de la primera línea de la tabla 9.1. Para estas especificaciones la **polaridad de la sincronización es negativa**, es decir, se pondrá un cero cuando haya sincronismo (como se ha dibujado en las figuras 9.3 y 9.4).

El bloque tendrá el aspecto de la figura 9.5.

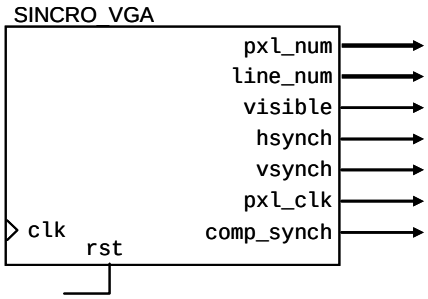


Figura 9.5: Entradas y salidas del sincronizador

Aunque las tarjetas *XUPV2P* y *Nexys2* tengan distintos conversores, nosotros usaremos el mismo bloque y si lo tenemos que implementar en la *Nexys2* simplemente dejaremos de usar los puertos que no sean necesarios. Las únicas diferencias importantes en la implementación es que funcionan a distinta frecuencia de reloj y la profundidad del color.

Aparte del reset y el reloj, la mayoría de las señales de la figura 9.5 están en la tabla 9.2 ó 9.3. La descripción de los puertos que no están en dichas tablas se muestra en la tabla 9.4.

Puerto	bits	I/O	Descripción
rst	1	I	Señal de reset asíncrono. Lo haremos activo a nivel bajo en la *XUPV2P* y a nivel alto en la *Nexys2*.
clk	1	I	Señal de reloj de la placa. 100 MHz en la *XUPV2P* y 50 MHz en la *Nexyx2*
visible	1	O	Indica con un uno si el módulo está enviando un píxel a la pantalla. Si es cero se está en el tiempo de sincronismo o porche delantero o trasero (figura 9.4)
pxl_num	10	O	Indica el píxel (columna), incluye todos los píxeles, tanto los visibles como los píxeles de la sincronización y los porches. Para una resolución de pantalla de 640x480 los valores irán de 0 a 800 (tomando la primera fila de la tabla 9.3).

`line_num`	10	O	Indica la línea (fila), incluye todas las líneas, tanto las visibles como las líneas de la sincronización y los porches. Para una resolución de pantalla de 640x480 los valores irán de 0 a 520 (tomando la primera fila de la tabla 9.3).

Tabla 9.4: Características de los puertos del módulo de sincronización. El resto de puertos están en las tablas 9.2 y 9.3

El módulo de sincronización se puede realizar de muchas maneras. Nuestra recomendación es hacerlo con tres contadores, que serían:

- Contador de ciclos de reloj: `P_cont_clk`. Cuenta los ciclos de reloj de cada píxel. Con este contador se pasa de la frecuencia del reloj de la FPGA a la frecuencia de cada píxel[71]. Con este método sólo el posible lograr las frecuencias que dividan (división entera) a la frecuencia de reloj.
- Contador de píxeles: `P_cont_pxl`. Cuenta los píxeles de cada línea.
- Contador de filas: `P_cont_line`. Cuenta las líneas de cada pantalla.

Te recomendamos que cada contador sea un proceso secuencial independiente en el que sólo se asigne la cuenta y que las señales que dependen de cada cuenta se asignen en un proceso combinacional aparte. Ya que si se ponen las señales dentro del proceso secuencial del contador se sintetizarán biestables y estarán retardadas respecto a la cuenta.

La figura 9.6 esquematiza el sincronizador. Como se puede observar, los procesos secuenciales únicamente llevan la cuenta, mientras que los procesos o sentencias combinacionales generan el resto de las señales. Ten cuidado de no generar *latches* en los procesos combinacionales (recuerda 2.5).

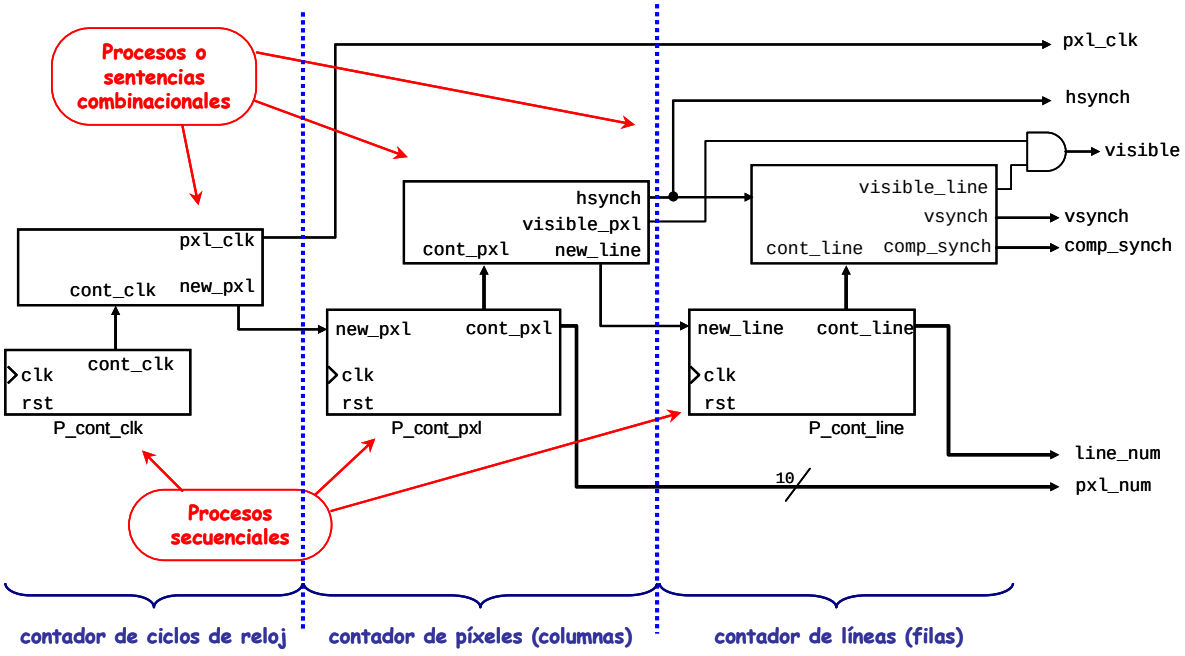


Figura 9.6: Esquema del sincronizador

El proceso contador de ciclos de reloj (`P_conta_clk`) será diferente si utilizamos la placa *XUPV2P* o la *Nexys2*, ya que sus relojes tienen distinta frecuencia y por lo tanto, la cuenta será distinta. Esto lo podemos ver en las siguientes figuras: la figura 9.7 muestra el

[71] indicado en la segunda columna de la tabla 9.3

cronograma de lo que queremos hacer si tenemos la *XUPV2P* y la figura 9.8 muestra el cronograma para la *Nexys2*.

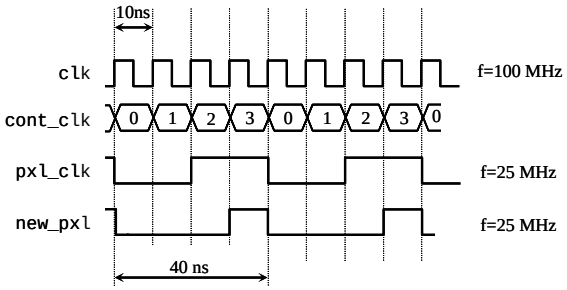


Figura 9.7: Cronograma del contador de ciclos de reloj para la XUPV2P (100 MHz)

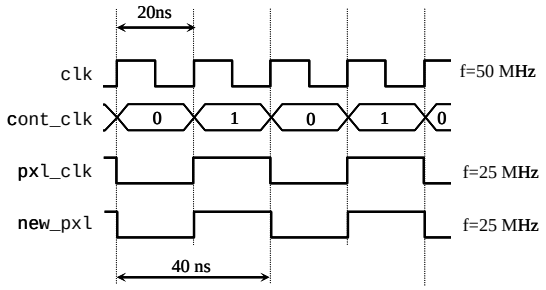


Figura 9.8: Cronograma del contador de ciclos de reloj para la Nexys2 (50 MHz)

Como la *XUPV2P* va a 100 MHz y queremos que enviar a la VGA los píxeles cada 25 MHz (recuerda la primera fila de la tabla 9.1), cada píxel durará 4 ciclos de reloj. Por otro lado, como la *Nexys2* va a 50 MHz, cada píxel se mantendrá durante 2 ciclos de reloj.

En los cronogramas se ha incluido una señal `new_pxl` que avisará al siguiente contador `P_cont_pxl` (que cuenta los píxeles) que aumente un píxel. Podemos observar que en el caso de la *Nexys2* esta señal coincide con `pxl_clk`. Aunque la *Nexys2* no necesita la señal `pxl_clk`, la mantendremos.

Así que el proceso que cuenta los ciclos de reloj (`P_cont_clk`) contará cuatro o dos ciclos según tengamos la tarjeta *XUPV2P* o la *Nexys2*. Lo ideal sería declarar el fin de cuenta mediante una constante en un paquete, y que al cambiar de tarjeta, sólo que tengamos que cambiar las constantes del paquete.

Por otro lado, como podemos ver en la tabla 9.1, tenemos que tener en cuenta muchos valores para el control de la VGA. Así que antes de empezar a describir el VHDL del sincronizador es conveniente declarar como constantes todos esos valores del control VGA que hemos escogido (primera fila de la tabla 9.1). Como vimos en el capítulo 6, el uso de constantes es muy recomendable. En nuestro caso, debido a que hay tantos valores distintos, se recomienda el uso de constantes por:

- Claridad: si en el código VHDL en vez de poner 640 ponemos `c_pxl_total`, será más fácil entender que nos referimos al número de píxeles totales. Un código más claro facilita la reutilización y disminuye la posibilidad de equivocarnos.
- Facilidad para cambiar: si queremos cambiar el valor de una constante, sólo tenemos que cambiar su valor en la declaración, en vez de buscarla por todos los ficheros en los que está. Esta última manera de hacerlo además de implicar mucho más esfuerzo en la búsqueda, favorece la aparición de errores ya que también hay constantes encubiertas. Es decir, habría que buscar y rehacer las operaciones con esas constantes. Por ejemplo, habría que buscar y recalcular un 639 que vendría de `c_pxl_total-1`.

Como ya vimos, para poder usar las constantes sin tener que redeclararlas en todos los ficheros VHDL, conviene poner todas las constantes de la VGA en un paquete. Este paquete lo podremos usar en todos los diseños de la VGA, y además nos será mucho más fácil de modificar el diseño si queremos cambiar la resolución de la pantalla.

En el código 9.1 se muestra cómo se podría hacer el paquete de la definición de las constantes. Estas constantes están basadas en la primera fila de la tabla 9.1.

En este paquete se utilizan funciones que ya hemos utilizado, como `log2i` (código 7.4). Por lo tanto habría que incluir otro paquete más general con estas funciones. No hace falta usar esta función y puedes asignar directamente el valor a las constantes, que en este caso es 10. Estos dos paquetes los puedes descargar en la dirección de la referencia [28web].

```
package VGA_PKG is
--  Orden:
--  1) video activo, 2) porche delantero, 3) sincro 4) porche trasero
--  Empezamos por el visible para que sea mas facil saber en que pixel visible
--  estamos, ya que si no, habria que realizar una resta
--  Por ejemplo para 640x480@60 Hz:
---------- Pixeles (horizontal, columnas) ---------------------------------|
-- |                         |             |                 |              |
-- |      video activo       |   porche    |     sincro      |    porche    |
-- |                         | delantero   |   horizontal    |    trasero   |
-- |                         |             |                 |              |
-- |<--------- 640 --------->|<--- 16 ---->|<------ 96 ----->|<---- 48 ---->|
-- |                         |             |                 |              |
-- |      c_pxl_visible      |c_pxl_fporch |  c_pxl_synch    | c_pxl_bporch |
-- |                         |             |                 |              |
-- |                                       |                 |              |
-- |<----- c_pxl_2_fporch: 656 ----------->|                 |              |
-- |                                       |                 |              |
-- |<------------- c_pxl_2_synch: 752 ---------------------->|              |
-- |                                                                        |
-- |<------------------------- c_pxl_total: 800 -------------------------->|

-- Pixeles (horizontal o columnas):
  constant   c_pxl_visible    : natural := 640;
  constant   c_pxl_fporch     : natural := 16;
  -- del inicio hasta el porche delantero:
  constant   c_pxl_2_fporch   : natural := c_pxl_visible + c_pxl_fporch; -- 656
  constant   c_pxl_synch      : natural := 96;
  -- del inicio hasta la sincronizacion:
  constant   c_pxl_2_synch    : natural := c_pxl_2_fporch + c_pxl_synch; -- 752
  -- total de pixeles horizontales:
  constant   c_pxl_total      : natural := 800;
  -- el porche trasero:
  constant   c_pxl_bporch     : natural := c_pxl_total - c_pxl_2_synch;  --  48

-- Filas (vertical):
  constant   c_line_visible    : natural := 480;
  constant   c_line_fporch     : natural := 9;
  constant   c_line_2_fporch   : natural := c_line_visible + c_line_fporch; -- 489
  constant   c_line_synch      : natural := 2;
  constant   c_line_2_synch    : natural := c_line_2_fporch + c_line_synch; -- 491
  constant   c_line_total      : natural := 520;
  constant   c_line_bporch     : natural := c_line_total - c_line_2_synch;  --  29

  -- numero de bits para la cuenta de pixeles(columnas) y lineas (filas)
  -- el logaritmo devuelve un bit menos, por eso se pone + 1
  -- y hay que ponerle un -1, porque en el caso de que la cuenta sea justo
  -- potencia de dos, vale con un bit menos, porque se cuenta el cero
  -- por ejemplo, 8: log2i(8)= 3 | + 1 = 4, pero con 3 bits, tengo de 0 a 7 (cuento 8)
  --              7: log2i(7)= 2 | + 1 = 3
  constant c_nb_pxls : natural := log2i(c_pxl_total-1) + 1 ;  --Para 640x480 es 10
  constant c_nb_lines: natural := log2i(c_line_total-1) + 1 ; --Para 640x480 es 10

  -- numero de bits para cada color (RGB) Para la XUPV2P
  constant c_nb_red   : natural := 8;  -- 3 para la Nexys
  constant c_nb_green : natural := 8;  -- 3 para la Nexys
  constant c_nb_blue  : natural := 8;  -- 2 para la Nexys

  -- frecuencia de la VGA
  constant c_freq_vga : natural := 25*10**6; -- VGA 25MHz

  -- nivel activo de sincronismo
  constant c_synch_act : std_logic := '0'; -- Para 640x480
end VGA_PKG;
```

Código 9.1: Paquete con la definición de las constantes del controlador VGA

Observa en el paquete del código 9.1 que la cuenta está referida a partir del video activo, en vez de empezar la línea con el sincronismo horizontal (como se mostraba en la figura

9.4). En realidad al monitor VGA le da igual por donde se empiece a contar, ya que son señales periódicas. Al monitor le es indiferente que lo primero que le enviemos sean los píxeles visibles o los de sincronismo. De hecho, probablemente la primera pantalla que enviemos no se muestre y el monitor necesite recibir varias señales de sincronismo antes de mostrar algo por pantalla. De la misma manera que podemos conectar un monitor a un ordenador en cualquier momento (y no justo cuando se envíe la señal de sincronismo), nosotros podemos empezar la cuenta por donde queramos siempre que mantengamos las frecuencias apropiadas.

Debido a que no es un problema donde se empieza la cuenta, recomendamos empezarla a partir de los píxeles visibles[72]. La ventaja de esto es que la cuenta se corresponderá con las coordenadas en la pantalla. Es decir, si la cuenta de píxeles (`pxl_num`) es 80 y la de líneas (`line_num`) es 40, se está dibujando en la pantalla el píxel de la columna 80 y de la fila 40([73]). Si comenzásemos la cuenta en otro punto, tendríamos que restar para saber la correspondencia entre la cuenta de píxeles y las coordenadas en la pantalla[74].

La figura 9.9 muestra el cronograma de la cuenta de píxeles cuando se empieza desde los píxeles visibles.

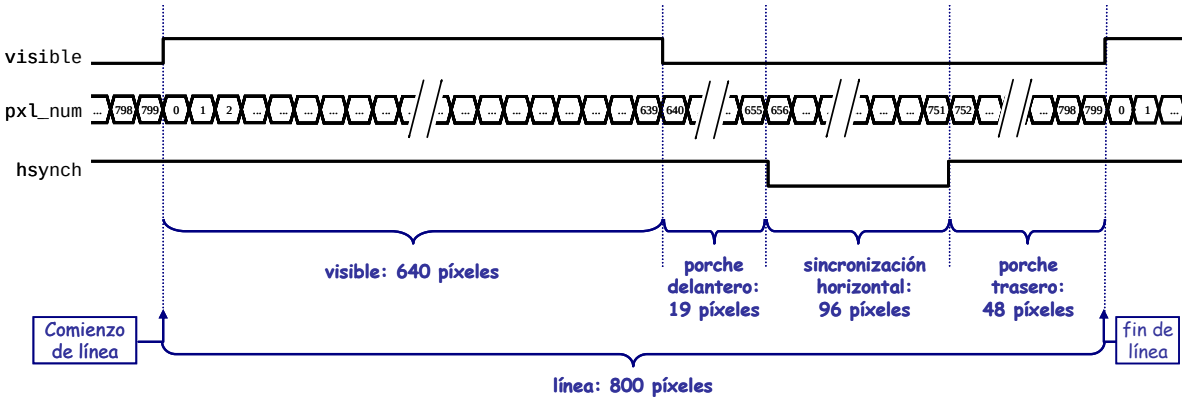


Figura 9.9: Cronograma de la cuenta de píxeles comenzando desde los píxeles visibles

Así pues, con las indicaciones que se han dado, crea el módulo `sincro_vga`. Una vez que lo tengas, simúlalo. La creación del banco de pruebas es sencilla, pues las únicas entradas son el reloj y el reset. La comprobación de la simulación es más difícil, pues hay que medir y comprobar que las cuentas y los tiempos se cumplen. En la figura 9.3 podemos ver que como la frecuencia de refresco es de 60 Hz, los sincronismos verticales deberían aparecer aproximadamente cada 17 ms. Por lo tanto, con 55 ms de simulación debería ser suficiente, para cubrir tres refrescos de pantalla. En la comprobación de la simulación deberás observar si se cumplen los tiempos de la figura 9.3, tanto de los píxeles como de las líneas.

9.4. Implementación de un controlador VGA sencillo

Una vez que tenemos el sincronizador, lo hemos simulado y hemos comprobado que las cuentas y tiempos son correctos, podemos pasar a probarlo con un monitor real.

[72] Y la cuenta de líneas a partir de las líneas visibles

[73] Considerando la columna cero la de la izquierda y la fila cero la de arriba (figura 9.2)

[74] Estas restas tampoco son un gran problema, pero al final el código queda menos claro y el diseño más sencillo

Para ello utilizaremos el sincronizador y lo incluiremos dentro de un módulo estructural de más alto nivel (`VGA_SIMPLE`). Dentro de este módulo estructural incluiremos un módulo (`PINTA_BARRAS`) que se encargará de dibujar unas barras de colores en el monitor que nos permitirán comprobar el correcto funcionamiento del sincronizador. El esquema estructural del diseño que queremos implementar se muestra en la figura 9.10.

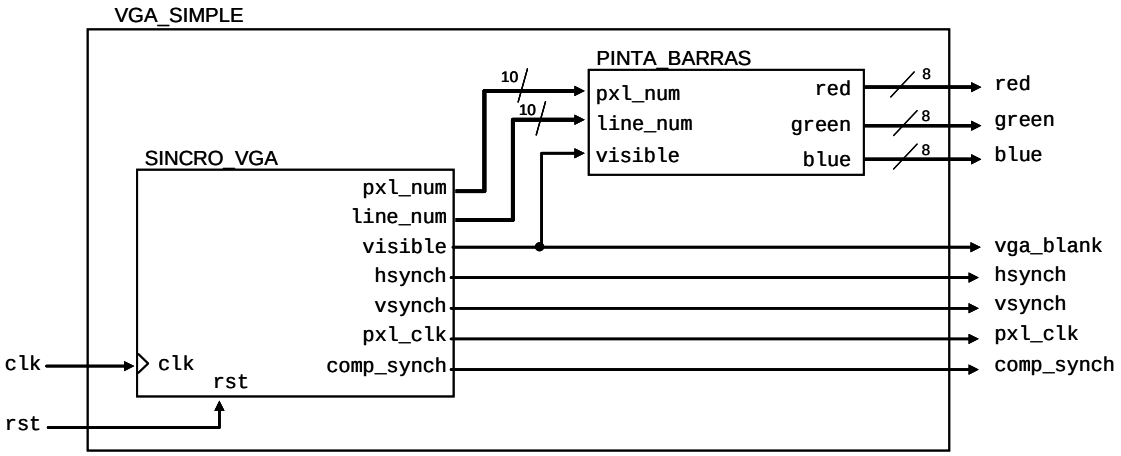


Figura 9.10: Esquema de bloques del controlador VGA sencillo

Las salidas son las que controlan la VGA y ya se explicaron en el apartado 9.2. Este esquema es para la *XUPV2P*, ya que la *Nexys* no necesita tantas señales y el ancho de bus de las señales del color es menor.

El módulo `PINTA_BARRAS` pondrá un color diferente según la columna de la pantalla en la que se esté. Como hemos puesto la cuenta de los píxeles y líneas coincidentes con las columnas y filas de la pantalla, no hay que hacer ninguna conversión. Cuando `visible` será cero, pondrá el color a negro (todo cero).

El código 9.2 muestra el código de la arquitectura del módulo `PINTA_BARRAS`. Este proceso está orientado para la *XUPV2P*, aunque funciona para la *Nexys2* si las constantes están bien definidas.

```
architecture behavioral of pinta_barras is
  constant c_bar_width : natural := 64;
begin
  P_pinta: Process (visible, pxl_num, line_num)
  begin
    red   <= (others=>'0');
    green <= (others=>'0');
    blue  <= (others=>'0');
    if visible = '1' then
      -- linea blanca de 1 pixel en los bordes
      if pxl_num = 0 OR pxl_num = c_pxl_visible -1  OR
        line_num=0   OR line_num= c_line_visible -1 then
        red   <= (others=>'1');
        green <= (others=>'1');
        blue  <= (others=>'1');
      elsif line_num >= 256 and line_num < 256 + c_bar_width then
        -- esto solo tiene sentido para la XUPV2P
        red   <= std_logic_vector(pxl_num(c_nb_red-1 downto 0));
        green <= std_logic_vector(pxl_num(c_nb_green-1 downto 0));
        blue  <= std_logic_vector(pxl_num(c_nb_blue-1 downto 0));
        if pxl_num >= 256 then
          red   <= std_logic_vector(pxl_num(c_nb_red-1 downto 0));
          green <= std_logic_vector(resize(255-pxl_num(7 downto 0),c_nb_green));
          blue  <= (others=>'0');
        end if;
        if pxl_num >= 512 then -- rayas horizontales
          red   <= (others=>not(line_num(0)));
          green <= (others=>not(line_num(0)));
          blue  <= (others=>not(line_num(0)));
```

```
        end if;
      elsif line_num >= 256+c_bar_width and line_num < 256 + 2*c_bar_width then
        red   <= std_logic_vector(resize(255 - pxl_num(7 downto 0),c_nb_red));
        green <= std_logic_vector(resize(255 - pxl_num(7 downto 0),c_nb_green));
        blue  <= std_logic_vector(resize(255 - pxl_num(7 downto 0),c_nb_blue));
        if pxl_num >= 256 then
          red  <=std_logic_vector(resize(255 - pxl_num(7 downto 0),c_nb_red));
          green<=(others=>'0');
          blue <= std_logic_vector(pxl_num(c_nb_blue-1 downto 0));
        end if;
        if pxl_num >= 512 then  -- puntos
          red   <= (others=>pxl_num(0) xor line_num(0));
          green <= (others=>pxl_num(0) xor line_num(0));
          blue  <= (others=>pxl_num(0) xor line_num(0));
        end if;
      elsif pxl_num/c_bar_width = 0 then
        -- columna 0 sera blanca, columna 1 negra, ...
        red   <= (others=>not(pxl_num(0)));
        green <= (others=>not(pxl_num(0)));
        blue  <= (others=>not(pxl_num(0)));
      elsif pxl_num/c_bar_width = 1 then     --blanco
        red   <= (others=>'1');
        green <= (others=>'1');
        blue  <= (others=>'1');
      elsif pxl_num/c_bar_width = 2 then     --amarillo
        red   <= (others=>'1');
        green <= (others=>'1');
        blue  <= (others=>'0');
      elsif pxl_num/c_bar_width = 3 then     --cian
        red   <= (others=>'0');
        green <= (others=>'1');
        blue  <= (others=>'1');
      elsif pxl_num/c_bar_width = 4 then     --verde
        red   <= (others=>'0');
        green <= (others=>'1');
        blue  <= (others=>'0');
      elsif pxl_num/c_bar_width = 5 then     --magenta
        red   <= (others=>'1');
        green <= (others=>'0');
        blue  <= (others=>'1');
      elsif pxl_num/c_bar_width = 6 then     --rojo
        red   <= (others=>'1');
        green <= (others=>'0');
        blue  <= (others=>'0');
      elsif pxl_num/c_bar_width = 7 then     --azul
        red   <= (others=>'0');
        green <= (others=>'0');
        blue  <= (others=>'1');
      elsif pxl_num/c_bar_width = 8 then     --negro
        red   <= (others=>'0');
        green <= (others=>'0');
        blue  <= (others=>'0');
      else         -- columna 639 sera blanca, 638 negra, ...
        red   <= (others=>pxl_num(0));
        green <= (others=>pxl_num(0));
        blue  <= (others=>pxl_num(0));
      end if;
    end if;
  end process;
end Behavioral;
```

Código 9.2: Proceso que asigna colores según el píxel

Lo que se muestra en la pantalla son unas barras verticales de colores a modo de carta de ajuste. La figura 9.11 muestra el resultado para la *XUPV2P*, como la *Nexys2* tiene menor profundidad de color, las barras horizontales son distintas[75] (sólo hay una barra horizontal).

[75] En la página web de la referencia [28web] puedes descargarte fichero más adecuado para la *Nexys2*.

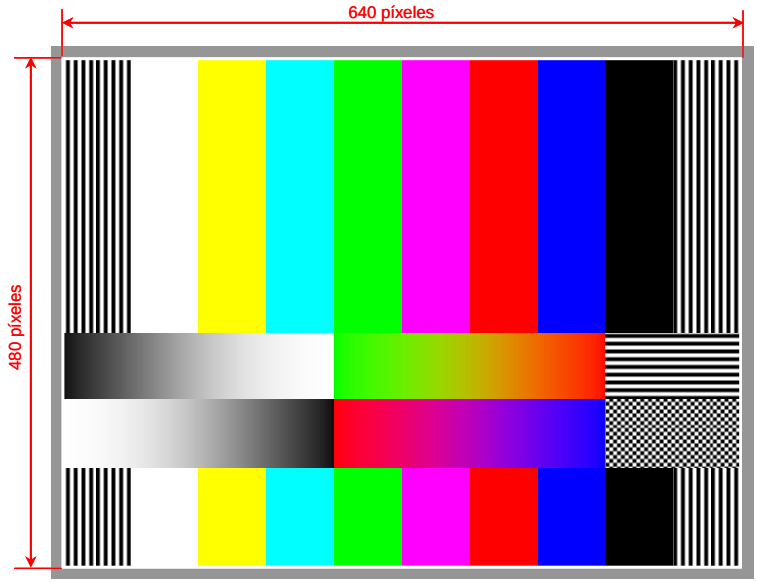


Figura 9.11: Carta de ajuste que debe mostrarse por pantalla para la XUPV2P

Si no consigues ver nada, probablemente el sincronismo esté mal. Aunque también puede ser por cualquier otro motivo, tendrás que hacer las comprobaciones que ya sabes: pines, conexiones de señales, cables,... Tendrás también que hacer un banco de pruebas para todo el diseño, comprobando que los puertos `red`, `green` y `blue`, se ponen en los valores fijados según en el píxel y la línea en la que se esté.

Una vez que consigas mostrar la carta de ajuste por la pantalla, debes comprobar que las barras se dibujan correctamente. Comprueba:

- Que se dibuja un marco de color blanco de un píxel de ancho en los cuatro bordes, muchas veces falta algún borde por pintar en blanco, o algunos de los bordes tiene más de un píxel de ancho. Como la resolución de la pantalla es baja, los píxeles se puede distinguir.
- Que las barras verticales tienen el mismo ancho
- Que las barras verticales de la izquierda y derecha están formadas por líneas alternativas de blanco y negro.
- Los colores de las ocho barras verticales: blanco, amarillo, cian, verde, magenta, rojo, azul y negro. Si el blanco no es blanco sino rosado puede ser porque no tienes la señal `comp_synch` correctamente (tabla 9.2), pero también puede ser que el monitor esté viejo. Comprueba que el monitor muestra los colores correctamente conectándolo a un ordenador[76]. Si los colores no se corresponden, comprueba que tienes los pines y las conexiones bien hechas.
- Que en las cuatro barras horizontales no hay saltos bruscos de color (si usas la *XUPV2P*). Si los hubiese, es posibles que algún puerto no esté conectado al pin adecuado.
- Las líneas horizontales blancas y negras de la barra horizontal de la derecha (la de arriba)
- Los píxeles blancos y negros alternados en la barra horizontal que está abajo a la derecha.

Haz los cambios para que se vea la carta de ajuste correctamente, quizá tengas que simular y ver los valores de los colores en las filas y columnas donde tengas los problemas

[76] También lo puedes comprobar si el monitor muestra una imagen cuando estando encendido no está conectado a ningún cable VGA

10. Videojuego de tenis

En esta práctica utilizaremos el controlador VGA de la práctica anterior para diseñar uno de los primeros videojuegos que se desarrollaron y se hizo muy popular a principios de los años setenta. Este videojuego está basado en el tenis de mesa (*ping pong*) y por esto fue comercializado con el nombre *Pong* por la empresa *Atari*.

En el videojuego hay un recuadro que marca los límites del campo de juego. Dentro de este campo hay dos barras verticales, una a la derecha y otra a la izquierda del campo, que representan a los dos jugadores. Hay una pelota que va de un jugador a otro (figura 10.1). La pelota puede rebotar en las paredes superior e inferior, pero si sobrepasa a los jugadores y alcanza las paredes verticales será gol.

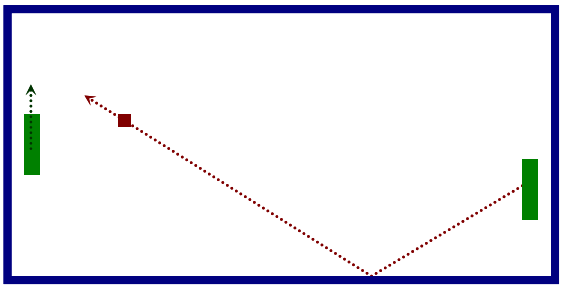


Figura 10.1: Campo de juego

Las barras verticales pueden moverse verticalmente dentro de los límites del campo de juego. En un principio, estas barras se mueven por la pulsación de los pulsadores de la placa.

Posteriormente se podrán controlar por el teclado PS/2 conectado a la placa (capítulo 11). Cada jugador tiene dos teclas: una para ir arriba y otra para ir hacia abajo. Si no se pulsa ninguna tecla o se pulsan ambas no habrá movimiento.

La figura 10.2 muestra un esquema de la implementación del videojuego. Este esquema tiene tres bloques:

- `SINCRO_VGA`: sincronizador de la VGA que hemos realizado en la práctica anterior
- `INTERFAZ_PB`: interfaz con los pulsadores de la placa
- `PONG`: encargado del control del juego, movimiento de pelota y jugadores, y de mostrarlo por pantalla.

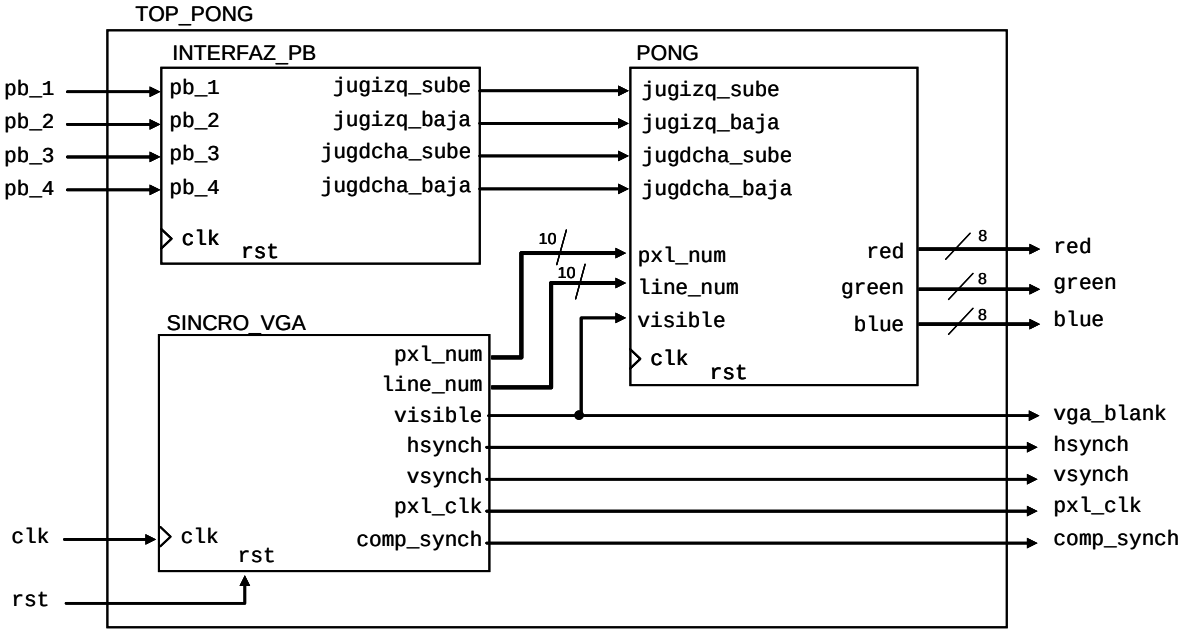


Figura 10.2: Esquema de bloques del juego de tenis controlado por los pulsadores

Como siempre, se recomienda realizar el diseño por pasos, de manera que podamos ir comprobando que todo va funcionando correctamente de manera independiente.

Así que los pasos podrán ser:

- Dibujar y hacer mover a un jugador por su zona a partir de la pulsación de sus botones.
- Incluir el segundo jugador y comprobar que se mueven de manera independiente
- Dibujar y mover una pelota de manera permanente por el campo de juego, sin interacción con los jugadores
- Incluir la interacción de los jugadores con la pelota e incluir el gol al alcanzar los bordes laterales

A continuación se darán algunas indicaciones para el diseño de estas etapas.

10.1. Control de los jugadores

Como hemos dicho, vamos a ir por partes en el diseño. Así que primero vamos delimitar el campo de juego. Este circuito es muy sencillo y ya hemos realizado algo similar. El diseño será más sencillo que la carta de ajuste que hicimos (figura 9.11). Pintaremos un campo de juego. Se recomienda limitar el campo de juego mediante constantes. De esta manera que podamos modificar su tamaño fácilmente. Así pues, la primera tarea será dibujar el campo de juego, puedes elegir los colores que quieras. Pinta los límites del campo de juego con más de un píxel de ancho, para que se vea bien. Realizaremos el diseño, lo implementaremos en la FPGA para asegurarnos que tenemos el primer paso correctamente.

Con el campo de juego definido, ahora dibujaremos los jugadores. Mejor empieza con un jugador, luego, cuando te funcione puedes pasar al otro.

En la figura 10.3 se muestran las dimensiones y coordenadas del jugador izquierdo respecto a la esquina superior izquierda. Esta esquina se considera el punto de coordenadas (0,0), y coincide con el píxel 0 de la línea 0 del sincronizador (figura 9.2).

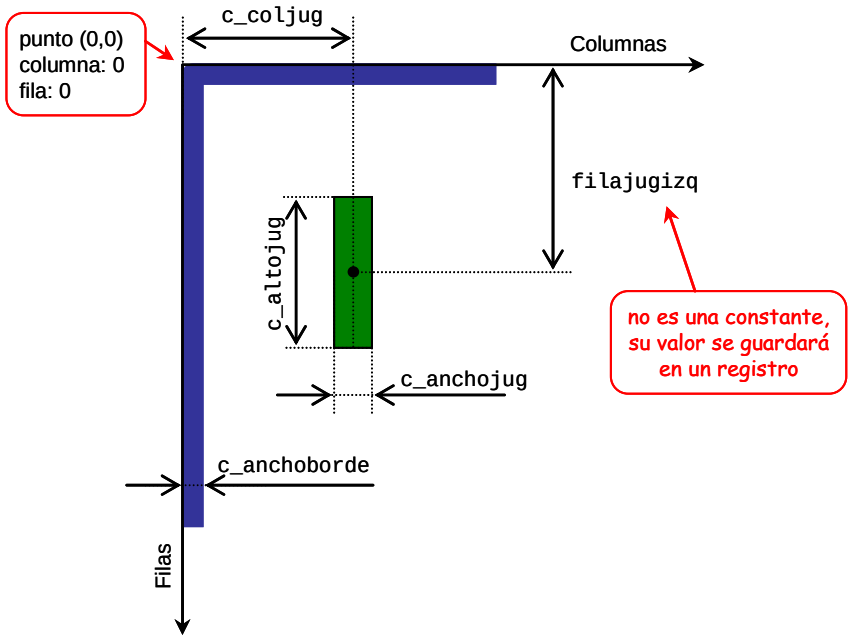


Figura 10.3: Coordenadas y medidas del jugador izquierdo en el campo

En la figura 10.3 todos los nombres que empiezan por "c_" indican constantes. Estas constantes conviene declararlas para que sea fácil de cambiar. Observa que la fila del jugador (`filajugizq`) no es una constante, sino que éste va a ser el valor que va a cambiar durante el juego ya que el jugador se podrá mover hacia arriba y abajo.

Se ha referido la posición del jugador con el registro `filajugizq` y la constante `c_coljug`. Estas coordenadas se han considerado como el centro del jugador (figura 10.3). Para calcular los cuatro bordes del jugador tendremos que sumar o restar a las coordenadas, las dimensiones divididas entre dos. Debido a esta división, es conveniente que las dimensiones del jugador sean un número par. Los bordes para el jugador izquierdo serán[77]:

- Fila del borde superior (variable): `filajugizq - c_altojug/2`
- Fila del borde inferior (variable): `filajugizq + c_altojug/2`
- Columna del borde izquierdo (constante): `c_coljug - c_anchojug/2`
- Columna del borde derecho (constante): `c_coljug + c_anchojug/2`

De estos bordes, como sólo se mueve verticalmente, las filas serán variables y las columnas constantes.

Con estos valores, antes de empezar a mover el jugador, dibuja un jugador en la pantalla poniendo su fila como una constante que luego cambiarás.

Una vez que tienes un jugador dibujado (o los dos), podemos empezar a moverlos. Primero debes de escoger los pulsadores de cada jugador y en qué sentido lo desplazan. Haz esto de modos que sean cómodos e intuitivos. Es decir, no elijas los pulsadores de la derecha para el jugador izquierdo, ni tampoco elijas como pulsador de un jugador uno que esté entre los pulsadores del otro jugador.

Tienes que definir el funcionamiento del movimiento de los jugadores en relación con los pulsadores. Es decir, podemos escoger entre estas dos opciones:

[77] En realidad, con estas fórmulas el ancho real es un píxel más ancho que la constante. Aunque luego depende de las condiciones y si se ponen comparaciones de mayor y menor, o mayor-igual y menor-igual.

- El jugador se mueve sólo al pulsar, pero si mantenemos pulsado el pulsador, no continúa el movimiento. Es decir tenemos que pulsar y soltar consecutivamente para mover el jugador.
- El jugador se mueve de manera continua mientras mantenemos el pulsador presionado

Para la primera opción tendríamos que realizar un detector de flanco. Recuerda el código 2.16 y la figura 2.13 en la que se puede ver cómo se transforma una pulsación larga en un único pulso de reloj[78]. Si lo hacemos de esta manera tendremos que pulsar muchas veces para mover el jugador y tendremos que variar en más de un píxel la posición del jugador, ya que de otra manera el movimiento será muy lento (píxel a píxel). Podemos tardar más de diez segundos en ir del borde superior al inferior.

Con la segunda opción se consigue un movimiento más rápido, pero tan rápido que hay que ralentizarlo. No podemos cambiar la posición del jugador cada ciclo de reloj en el que tengamos presionado el pulsador. Si lo hacemos así, el jugador irá tan rápido que ni lo veremos, y pasará de estar en un extremo al otro (eso si están bien los controles del jugador para que se mantenga dentro del campo de juego). Así que para este caso la solución está en muestrear la señal cada cierto tiempo (entre 5 y 10 ms son valores razonables). Tendremos que realizar un contador que al terminar su cuenta envíe una señal de aviso para que se comprueben las señales de los pulsadores, si alguno está activado, deberá enviar un pulso[79] de un ciclo de reloj al módulo `PONG` para que aumente o disminuya en una línea la posición del jugador. Es importante volver poner las señales a cero después de muestrear, porque si no, de nuevo irá muy rápido el jugador.

Esta segunda opción queda mejor y no es más difícil de hacer que la primera.

Ten en cuenta ninguna parte del jugador debe sobrepasar los límites del campo, por tanto, para controlar la posición del jugador debes tener en cuenta su altura y dónde están los bordes. Por ejemplo, se debe de impedir que suba el jugador cuando está en el borde superior.

En el caso de que el movimiento sea con los flancos hay que tener más cuidado en el control de que el jugador no se salga del campo porque el movimiento es en saltos de varias líneas, y puede ser que "salte" fuera del campo.

10.2. Pelota que rebota

Una vez que tenemos a los jugadores vamos a incluir la pelota. En la primera versión la pelota y los jugadores van a ser independientes. La pelota rebotará en las cuatro paredes del campo de juego y no chocará con los jugadores. La pelota será cuadrada y tendrá un ancho par definido en las constantes del módulo (`c_anchobola`).

La pelota puede variar tanto su fila como columna, no como los jugadores que sólo podían variar su fila. Por lo tanto se necesitan dos registros para guardar sus coordenadas. Pero también necesitamos un registro que guarde la dirección del movimiento de la pelota.

[78] También hicimos un detector de flanco en el transmisor, recuerda la figura 7.27

[79] Estas señales son: `jugizq_sube`, `jugizq_baja`, `jugdcha_sube` y `jugdcha_baja`

Para simplificar, limitaremos el movimiento de la pelota a las cuatro direcciones mostradas en la figura 10.4. Estas direcciones las llamaremos `NO`, `NE`, `SE` y `SO`, que corresponden con las direcciones noroeste, noreste, sureste y suroeste respectivamente. En VHDL, estas cuatro direcciones las podemos declarar como un enumerado.

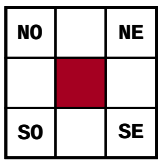


Figura 10.4: cuatro direcciones de la pelota

La pelota seguirá la misma trayectoria hasta que alcance uno de los bordes. Al llegar a éstos cambiará la dirección tomando la dirección simétrica. La figura 10.5 muestra dos ejemplos de cambios de trayectoria de la pelota. El resto de choques son fácilmente deducibles a partir de éstos.

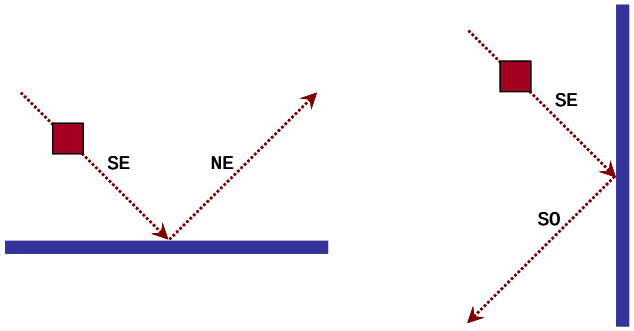


Figura 10.5: Ejemplos de cambios de trayectoria de la pelota al chocar con las paredes

Igual que en el caso de los jugadores, la posición de la pelota no puede cambiar cada ciclo de reloj. Tendremos que crear un contador de modo que cada cierto tiempo su posición varíe. Este tiempo puede ser del mismo orden de magnitud que el de muestreo de los jugadores. Cada vez que el contador avise que se llega al fin de cuenta, se debe comprobar que, siguiendo su trayectoria, la pelota no choca con los bordes. Si la pelota no choca se cambiará su fila y columna en un píxel según la dirección indicada por la trayectoria. En el caso de haya choque, la pelota cambiará su trayectoria, y el cambio de fila y columna seguirá la nueva trayectoria.

Puede pasar que cuando la pelota choca, si no cambias la fila y columna de manera consecuente con la nueva trayectoria, la pelota se quede enganchada en un borde. En este caso debes revisar las condiciones de choque y asegurarte que en el momento del choque se cambien la fila y columna según la nueva trayectoria y no con la trayectoria anterior al choque.

Al principio y después de cada gol, la pelota puede salir del centro del campo, con una dirección pseudo-aleatoria. Para conseguir que sea lo más aleatorio posible hay que hacer depender el valor de algo aleatorio, por ejemplo, un contador que dependa de la última vez que se ha empezado a pulsar cualquiera de los pulsadores.

10.3. Videojuego

Uniendo los jugadores y la pelota construiremos la primera versión del videojuego. Ahora la pelota estará limitada por los jugadores y no por los límites verticales (izquierdo y derecho) del campo. Si la pelota llega a uno de los bordes verticales del campo de juego significa que el jugador contrario ha metido gol.

Este diseño se puede mejorar en muchos aspectos:

- Aumentar el número de direcciones y añadir efectos, por ejemplo, si choca con un jugador que tiene velocidad
- Variar la velocidad de la pelota.
- Añadir un marcador que cuente los goles de cada jugador. Si quieres hacer esto sería mejor que esperes a ver el capítulo de pintar caracteres por pantalla (capítulo 13). Aunque no está mal que lo pienses antes de ver el capítulo.
- Mover los jugadores con un teclado PS/2, lo veremos en el capítulo siguiente.
- Poner una pelota redonda y jugadores menos rudimentarios. Lo veremos en el capítulo de dibujar una imagen por pantalla (capítulo 12).

Y cualquier otra idea que se te ocurra, sin embargo, a no ser que tengas un determinado interés, no es necesario hacer mejoras que no impliquen ningún aprendizaje nuevo. Para el objetivo de aprendizaje de este libro, no tiene mucho sentido mejorar el diseño a base de añadir muchas más condiciones en los procesos sin más.

11. Puerto PS/2

El puerto PS/2 se utiliza para conectar el teclado y el ratón al ordenador. Aunque cada vez se usa menos desde la aparición de los teclados y ratones USB, todavía se ven muchos ordenadores con estos conectores.

El conector PS/2 consta de 6 pines, que es de tipo mini-DIN (figura 11.1). El conector del teclado puede tener un color violeta o morado y el del ratón puede ser verde, aunque a veces no tienen un color específico.

El puerto PS/2 consta de cuatro cables: la alimentación (4,5-5,5V), la tierra, el dato y el reloj. La comunicación se realiza con las líneas de dato y reloj. La comunicación se realiza entre un **dispositivo**: el teclado o el ratón, y la **computadora** o la FPGA (el *host*).

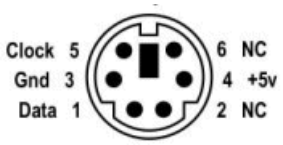


Figura 11.1: Conector PS/2

El teclado y ratón PS/2 implementan un protocolo serie bidireccional síncrono con la computadora. El bus está en espera (*idle*) cuando ambas líneas (dato y reloj) están a uno. La computadora tiene el control último sobre el bus, pudiendo inhibir la comunicación en cualquier momento poniendo un cero en la línea de reloj.

El teclado puede funcionar de forma unidireccional, de modo que la computadora simplemente reciba las tramas del teclado. Sin embargo, para el funcionamiento del ratón se requiere un modo bidireccional. Por tanto, el funcionamiento del teclado es más sencillo que el del ratón y por esto se explicará antes.

Atención: No se recomienda conectar un dispositivo PS/2 con el ordenador encendido. Por si acaso, te recomendamos que sigas la misma recomendación cuando conectes el dispositivo a la placa de la FPGA, es decir, que los conectes con la placa apagada.

11.1. Teclado PS/2

Como sólo necesitamos recibir las tramas del teclado PS/2, realizaremos un receptor de teclado PS/2, dejando el transmisor para el ratón PS/2.

Las tramas enviadas por el protocolo PS/2 son de 11 bits, teniendo un bit de inicio, 8 bits de dato, un bit de paridad impar y un bit de fin. De manera similar al protocolo RS232, la línea de datos se mantiene a valor lógico '1' cuando está inactiva, siendo el bit de inicio un cero y el bit de fin un uno. Los ocho bits de datos se envían empezando por el menos significativo. El bit de paridad impar estará a uno si hay un número par de unos en los 8 bits de datos, y estará a cero si hay un número impar. Por tanto, contando los 8 bits de datos más el bit de paridad, deberá haber un número impar de unos (por eso es paridad impar). Esto se utiliza para detectar errores. La figura 11.2 muestra el cronograma de un envío del teclado a la computadora.

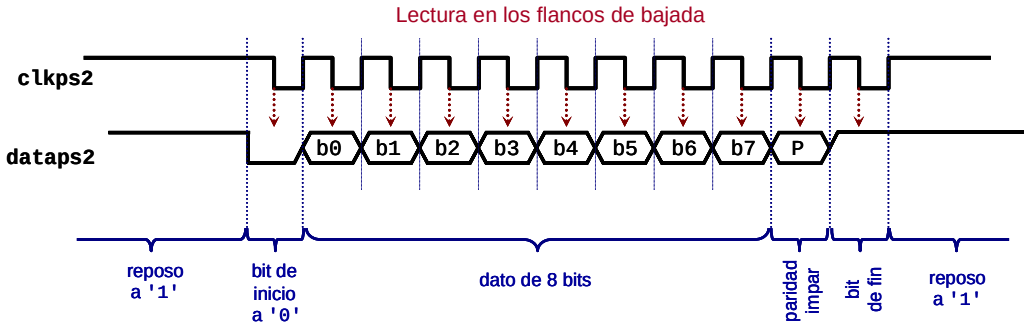


Figura 11.2: Cronograma general de la transmisión del puerto PS/2

Podemos ver que la transmisión es muy parecida a la de la UART, la diferencia más importante es que el PS/2 tiene dos líneas: la de datos (`dataps2`) y la del reloj (`clkps2`). La línea del reloj es la que marca la frecuencia, indicando los cambios de bit y cuándo se debe de leer. La frecuencia del reloj es de un rango entre 10 y 16,7 kHz (entre 60 y 100 µs). La lectura del bit se debe realizar en los flancos de bajada del reloj del PS/2 (`clkps2`).

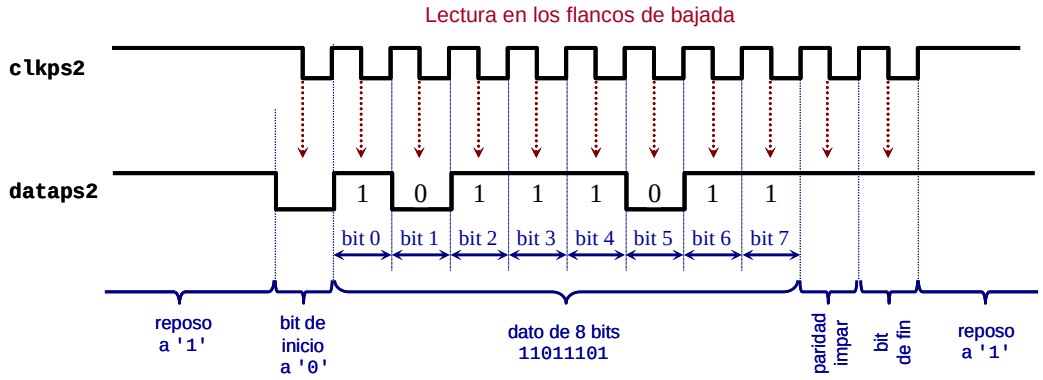


Figura 11.3: Ejemplo del cronograma para una transmisión del teclado

Con todo esto, podemos deducir que el diseño del receptor del teclado PS/2 será muy parecido al receptor de la UART. La mayor diferencia está en la generación de la señal para la lectura de los bits. Recuerda que en el receptor de la UART, para leer los bits de la trama teníamos que generar la señal `baud_medio` a partir del divisor de frecuencia (apartado 7.7.3 y figura 7.48). Ahora ya no tenemos que generar esa señal porque la lectura viene marcada por los flancos de bajada de la señal de reloj del PS/2 (`clkps2`). Lo que tenemos que hacer es un detector de flancos de bajada de la señal `clkps2`. Tendremos que leer cada bit de la trama cada vez que se detecta un flanco de bajada.

Así pues, la entidad del receptor del teclado podrá ser como la mostrada en el bloque de la figura 11.4 . Observa que este receptor de se parece mucho al de la UART (figura 7.45). La diferencia es que ahora se tiene el puerto `ps2clk` que es el reloj de la transmisión que viene del teclado. También que se ha añadido la señal de error (`error_ps`) que indica si ha habido error en la transmisión, calculado a partir de la trama recibida y su bit de paridad.

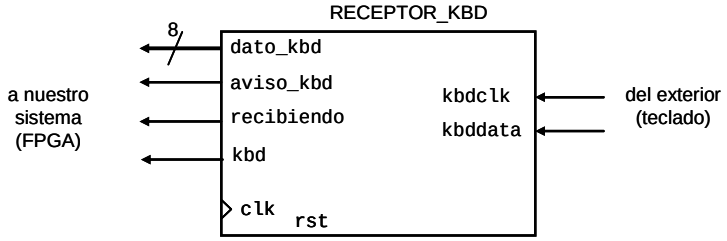


Figura 11.4: Entradas y salidas del receptor del teclado

En principio, con lo que hemos estudiado hasta ahora, no deberías de tener muchos problemas para realizar este diseño.

Se recomienda incluir uno o dos registros a la entrada de las señales externas (`ps2clk` y `ps2data`) para evitar metaestabilidad, y hacer las lecturas y la detección de flanco con las señales registradas. La metaestabilidad se explicó en el capítulo 5 de la referencia [17mach].

11.1.1. Códigos *scan*

Hemos visto cómo recibir la información del teclado, pero quizá te hayas preguntado qué significado tienen las tramas recibidas. Es decir, cuál es la relación entre las teclas que pulsamos y lo que se envía por el PS/2. La relación viene determinada por los códigos *scan*. La figura 11.5 muestra los códigos *scan* de las teclas de un teclado inglés norteamericano.

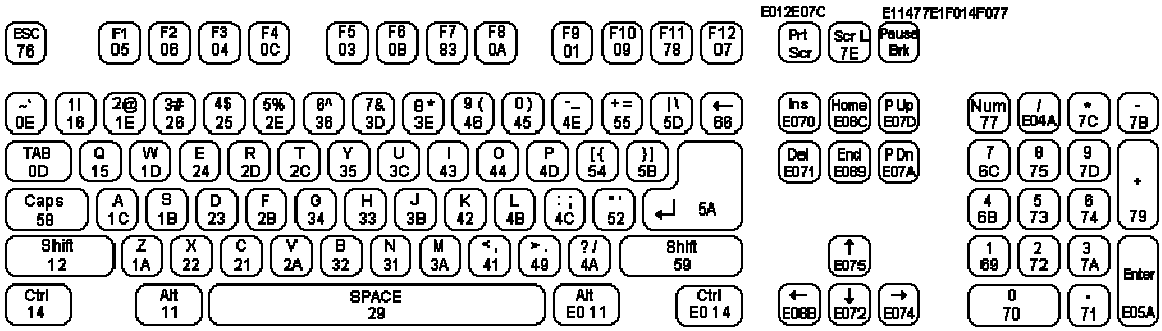


Figura 11.5: Códigos scan de un teclado inglés norteamericano

Los códigos *scan* de la figura 11.5 están en hexadecimal, la mayoría de ellos tiene dos dígitos hexadecimales, por lo tanto se pueden enviar en los 8 bits de datos de la trama del PS/2. Sin embargo puedes observar que hay caracteres especiales que tienen más de un código *scan* y que su código está precedido por E0h. Por ejemplo, la flecha hacia arriba tiene el código E0h - 75h. Estos códigos *scan* dobles se llaman códigos *escapados* (*escaped scancodes*) o teclas extendidas. Puedes ver que incluso hay algunos que tienen más de dos. Normalmente los códigos *escapados* tienen alguna relación con los sencillos. Por ejemplo la tecla de control *Ctrl* de la izquierda tiene el código 14h, y la de la derecha es E0h - 14h.

Los códigos *scan* que envía el teclado son de dos tipos: *make* y *break*. Cada vez que se pulsa una tecla se envía un código *make* y cuando se deja de presionar se envía un código *break*. El código *break* es igual que el *make* pero precedido por el número F0h (en hexadecimal). Con esto se puede saber si se está pulsando una tecla mientras hay otra tecla pulsada, lo que es útil por ejemplo para las teclas de mayúsculas y control: *Shift*, *Alt*, *Ctrl*,....

El teclado no realiza ninguna interpretación de las teclas pulsadas, esto lo tiene que realizar la computadora (o en nuestro caso, la FPGA). Es decir, que si mantenemos presionada la tecla *Shift* cuando pulsamos la letra "*a*", el teclado no envía la letra "*A*" mayúscula, sino que le corresponde a la computadora interpretar que se ha enviado letra "*A*" la mayúscula.

En los códigos *scan* escapados, cuando se envía el código *break*, se envía entre los dos códigos *scan*. Por ejemplo, el código *break* de la flecha hacia arriba será E0h - F0h - 75h.

Por poner un ejemplo, si queremos escribir la letra Z mayúscula, sucederá lo siguiente:

- Presionamos la tecla *Shift*: 12h, y sin soltarla
- Presionamos la tecla Z: 1Ah
- Soltamos la letra Z: F0h - 1Ah
- Soltamos la tecla *Shift*: F0h - 12h

Así que el teclado enviará: 12h - 1Ah - F0h - 1Ah - F0h - 12h.

Si una tecla se mantiene pulsada se convierte en *typematic*, que significa que el teclado reenviará su código *make* aproximadamente cada 100 ms, hasta que se suelte o hasta que se presione otra tecla. Para verificar esto puedes abrir un editor de texto y hacer la prueba: Pulsa una tecla, por ejemplo la "B", e inmediatamente aparecerá la letra "b" en el editor. Si la has mantenido pulsada, al cabo de un breve espacio de tiempo volverá a aparecer repetidamente hasta que la sueltes o hasta que pulses otra tecla. En este último caso, esta última se convertirá en *typematic*. Esto hay que tenerlo en cuenta en el control, cuando se espera que se mantengan presionadas varias teclas simultáneamente.

Con esto ya tenemos información suficiente para hacer un receptor del teclado y, si fuese necesario, un control que gestione las teclas que se pulsan.

Para probar este diseño puedes probar a enviar por la UART lo que se recibe por el teclado o puedes incluir el control del videojuego *Pong* por el teclado.

11.2. Ratón PS/2

El interfaz con el ratón es más complicado porque el ratón necesita recibir información desde la computadora para empezara funcionar, mientras que para el teclado no es necesario aunque puede recibir información desde la computadora para su configuración. En el apartado anterior hemos visto cómo recibir datos del teclado sin necesidad de enviar al teclado ninguna información.

En este apartado vamos a introducir un concepto muy interesante que son los puertos bidireccionales. Esto es, puertos que pueden enviar y recibir datos.

En este apartado vamos a ver:

- Funcionamiento del ratón
- Protocolo PS/2 para enviar información al ratón desde la computadora
- Puertos bidireccionales
- Descripción de puertos bidireccionales en VHDL

11.2.1. Funcionamiento del ratón

El interfaz estándar PS/2 del ratón admite las siguientes entradas:

- Movimiento en el eje X (izquierda/derecha)

- Movimiento en el eje Y (arriba/abajo)
- Botón izquierdo
- Botón central
- Botón derecho

El ratón lee estas entradas a una frecuencia determinada y actualiza los contadores e indicadores para reflejar los estados de movimiento y botones. Existen muchos dispositivos apuntadores que tienen entradas adicionales y pueden proporcionar datos de manera diferente a la que se especifica aquí (como por ejemplo los que tienen la ruedecita de *scroll*).

El ratón estándar tiene dos contadores que recogen la información del movimiento: los contadores de movimiento en X y en Y. Estos valores se codifican con 9 bits en complemento a 2, teniendo cada uno asociada una bandera de desbordamiento. El contenido de estos contadores, así como el estado de los tres botones, se envían a la computadora en un paquete de tres bytes. Los contadores de movimiento representan el movimiento que el ratón ha experimentado desde que se envió el último paquete, por tanto no representan posiciones absolutas.

El contenido de estos bytes se muestra en la figura 11.6. El protocolo de envío es el explicado anteriormente en la figura 11.2. En la figura 11.6 sólo se muestran los ocho bits de datos de los tres envíos y no se muestran los bits de incio, paridad y fin.

Figura 11.6: Contenido de los tres bytes de información enviados por el ratón

El ratón se puede configurar para cambiar parámetros como la velocidad de muestreo, la resolución, el escalado y si está habilitada la transmisión de datos o no.

Hasta aquí el funcionamiento es similar al del teclado, sin embargo este funcionamiento corresponde con el modo de funcionamiento *stream* (transmisión) cuando está habilitada la transmisión de datos, pero no es el modo de funcionamiento con el que arranca el ratón.

El ratón tiene varios modos de funcionamiento: reset, transmisión (*stream*), remoto y eco (*wrap*). Los modos remoto y eco casi no se usan, así que sólo vamos a ver los dos primeros.

El **modo reset** es el modo en el que entra el ratón al encenderse. Al entrar en este modo el ratón realiza un test de autodiagnóstico y pone los siguientes parámetros:

- Velocidad de muestreo: 100 muestras/s
- Resolución: 4 cuentas por milímetro (por cada milímetro los contadores aumentan en 4)
- Escalado: 1:1
- Envío de datos: deshabilitado

Una vez que el ratón realiza el autodiagnóstico envía un código que indicará si ha tenido éxito (AAh) o si ha habido algún error (FCh). Luego enviará la identificación del dispositivo PS/2, que para un ratón PS/2 estándar es 00h.

Una vez que el ratón ha enviado la identificación del dispositivo entra en el **modo de transmisión**.

En el modo de transmisión el ratón envía la información si tiene habilitada esta función. Sin embargo, hemos visto que al arrancar el ratón en el modo reset, se deshabilita esta función.

Para habilitar la transmisión, la computadora tiene que enviar el comando F4h (*Enable Data Reporting*). Una vez que el ratón recibe este comando, éste responderá con el comando FAh para indicar el reconocimiento del comando.

A partir de ahora el ratón enviará datos normalmente y ya no será necesario enviar más comandos al ratón desde la computadora.

Se ha incluido la tabla 11.1 para resumir los comandos que se envían entre el ratón y la computadora (o FPGA) al arrancar,

secuencia en el tiempo	1	2	3	4	5	...
ratón → FPGA	AAh	00h		FAh	informe de movimiento del ratón (figura 11.6.)	
FPGA → ratón			F4h			

Tabla 11.1: Comunicación entre el ratón y la FPGA hasta habilitar la transmisión de datos del ratón

Si queremos hacer un sistema lo más sencillo posible en nuestra FPGA, nos bastaría con enviar el comando F4h pasado un tiempo desde el arranque y esperar a recibir el comando de reconocimiento FAh, una vez hecho esto, el sistema pasaría a recibir la información del movimiento del ratón. Por otro lado, hay que tener en cuenta que si estamos configurando la FPGA desde el cable JTAG y el ratón está conectado[80], el ratón habrá enviado estos dos comandos antes de que la FPGA esté configurada. Por tanto, si ya los ha enviado, nuestro sistema se podría quedar bloqueado esperando a recibirlos.

Otra alternativa es forzar el reset enviando el comando de reset FFh, tras este comando, el ratón enviará el reconocimiento y comenzará con la secuencia de la tabla 11.1.

A continuación veremos cómo funciona el protocolo PS/2 para enviar un comando desde la computadora al ratón

11.2.2. Protocolo PS/2 para transmitir desde la computadora

Normalmente el control de la línea de datos y del reloj PS/2 lo tiene el dispositivo (ratón o teclado). Si la computadora (o FPGA) quiere tomar el control debe de poner la línea del reloj (`clkps2`) a cero durante 100 μs, después de este tiempo, libera la línea del reloj y pone la línea de datos a cero (bit de inicio). Posteriormente, con la línea de reloj controlada ya por el dispositivo, espera a cada flanco de bajada del reloj para poner cada uno de los datos y el bit de paridad. Finalmente, la computadora libera la línea de datos con el bit de fin y el dispositivo pone un cero de reconocimiento en la línea de datos.

La figura 11.7 muestra el cronograma de las líneas de reloj y datos del PS/2 cuando la computadora (o FPGA) quiere enviar datos al dispositivo. Como la línea es bidireccional, en la figura se ha dibujado en rojo los valores puestos por la computadora y en azul los valores puestos por el ratón.

[80] Recuerda que no es recomendable conectar el ratón o el teclado PS/2 con la placa de la FPGA encendida.

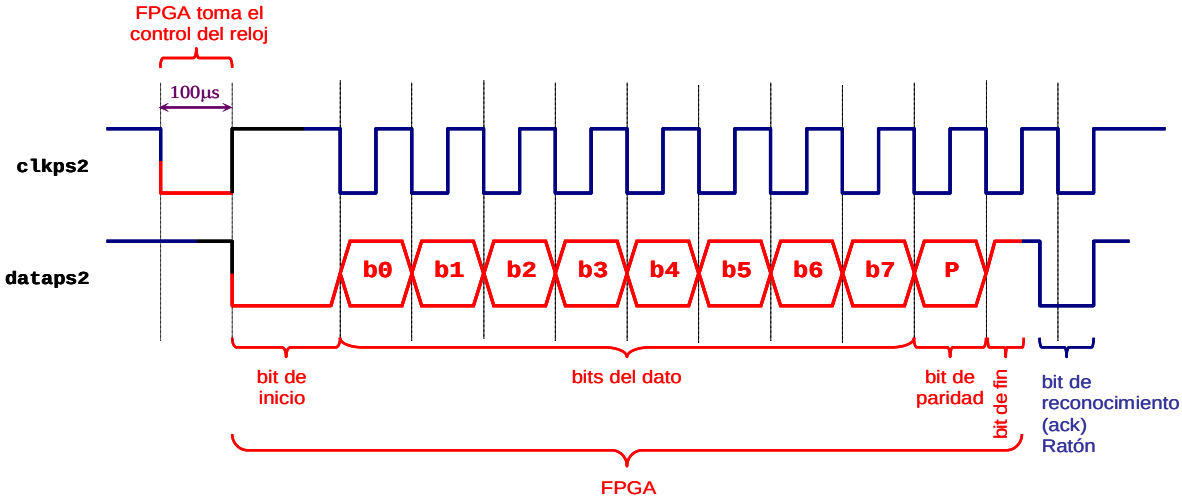


Figura 11.7: Cronograma de la transmisión de datos entre la computadora o FPGA (rojo) y el ratón (azul)

Para clarificar más el cronograma, en la figura 11.8 se han separado los cronogramas según quién pone el valor. En el de arriba y en rojo se han puesto los valores puestos por la computadora y abajo en azul las del ratón. Para ver mejor las relaciones entre las señales, en cada cronograma se incluye en línea punteada los valores de la señal que son puestos por el otro dispositivo.

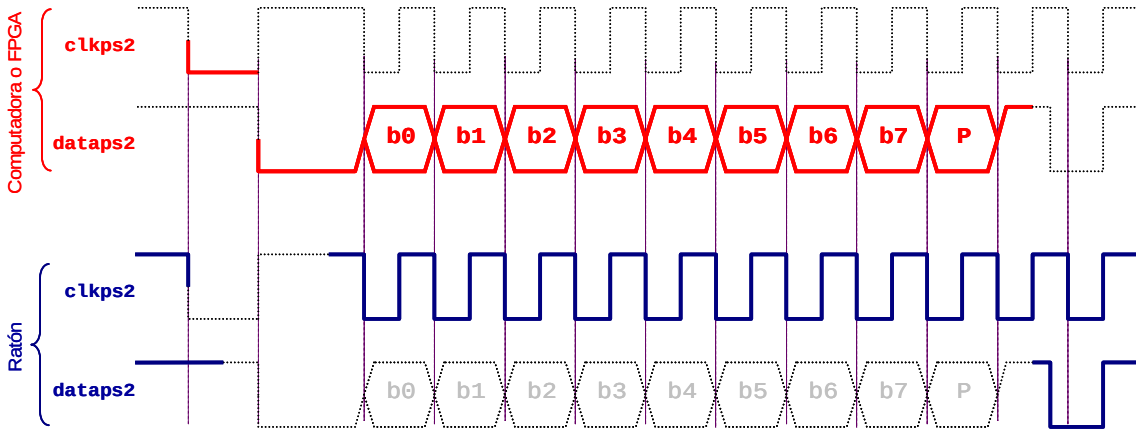


Figura 11.8: Cronogramas separados de la transmisión de datos desde la computadora hacia el ratón

Para mayor claridad, a continuación se explica con más detalle el cronograma:

- La computadora toma el control de la línea del reloj poniéndola a cero durante al menos 100 µs. Con esto se inhibe la actividad del ratón. Esto se puede interpretar como que la computadora solicita realizar un envío
- Pasados 100 µs, la computadora libera la línea de reloj y pone un cero en la línea de datos. Este cero se puede considerar como el bit de inicio.
- La computadora mantiene el cero en la línea de datos hasta que el ratón toma el control de la línea de reloj y la pone a cero. Esta transición de uno a cero (flanco de bajada) se puede considerar como un reconocimiento por parte del ratón de que ha recibido el bit de inicio, y el indica a la computadora que envíe el primer bit del dato (el bit 0)
- La computadora pone el bit número cero cuando detecta el flanco de bajada del reloj puesto por el ratón. El ratón indica con cada flanco de bajada del reloj que está preparado para recibir el siguiente bit. Por lo tanto, cada vez que la computadora detecta

un nuevo flanco de bajada del reloj pondrá un nuevo bit. Esta operación se repite con todos los bits del dato hasta llegar al bit de paridad.

- La computadora, una vez que detecta el flanco de bajada de reloj después del bit de paridad, libera la línea de datos, lo que hace que se ponga a uno. Esto sería el bit de fin.
- Una vez liberada la línea de datos por la computadora, el ratón pondrá la línea de datos a cero, indicando que ha reconocido la transición[81].
- La computadora podrá verificar el bit de reconocimiento en la línea de datos cuando detecte un nuevo flanco de bajada de la línea de reloj[82].
- Para terminar, el dispositivo liberará la línea de datos y de reloj, poniéndose ambas a uno

Este protocolo es válido tanto para el ratón como para el teclado, pero para el teclado no lo hemos necesitamos porque por ahora no hemos querido enviarle ningún comando.

Con esta información y con la del apartado anterior ya podríamos proceder a realizar el interfaz con el ratón.

Sin embargo, puede que te hayas estado preguntando, cómo puede haber una línea bidireccional. Hasta ahora hemos seguido la regla de que una señal no puede ser asignada desde dos sitios diferentes y hemos usado puertos que son o bien de entrada o bien de salida.

A continuación veremos cómo funcionan y cómo se implementan los puertos bidireccionales.

11.2.3. Puertos bidireccionales

Las líneas de reloj y dato del PS/2 son en drenador abierto[83], esto permite realizar conexiones bidireccionales sobre el mismo cable. La figura 11.9 muestra un esquema muy simplificado de la conexión en drenador abierto entre la FPGA y el ratón. El esquema está muy simplificado y sólo pretende explicar el concepto, además de que el esquema real es diferente para la *XUPV2P* y la *Nexys2*.

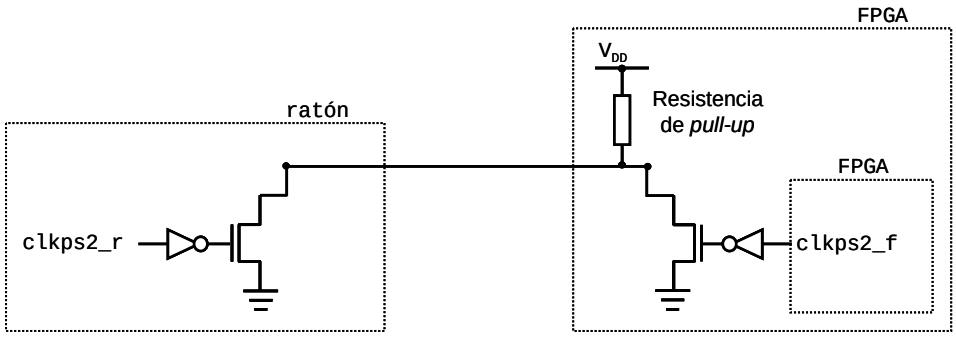


Figura 11.9: Esquema simplificado de la conexión en drenador abierto delPS/2

La figura 11.10 esquematiza el funcionamiento de un transistor en drenador abierto conectado a una resistencia de *pull-up*. El inversor que se ha colocado después de la señal `A` es para que la salida `Linea` tenga el mismo valor, si no, `Linea` estaría invertida.

[81] En inglés se llama *acknowledge bit* (`ack`)

[82] Este reconocimiento no es necesario implementarlo en la FPGA

[83] O colector abierto en caso de que sea con transistores bipolares en vez de con MOSFET

La resistencia de *pull-up* hace que la salida `Linea` esté a V_{DD} cuando el transistor no conduce. Cuando el transistor conduce, la salida `Linea` tiene una tensión casi cero.

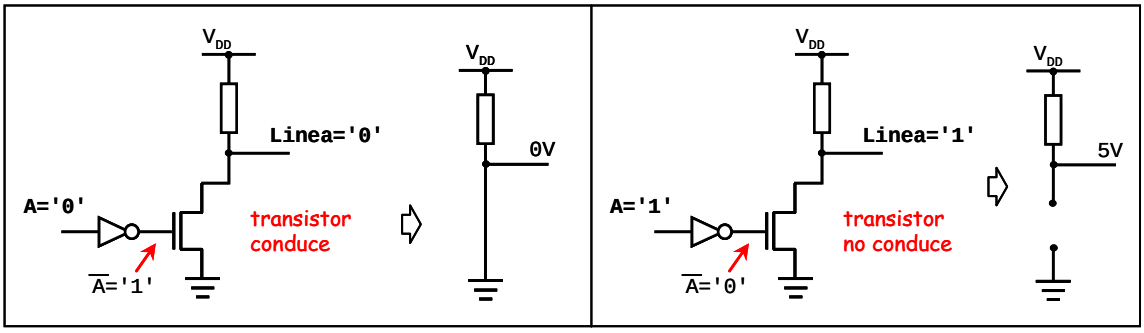


Figura 11.10: Funcionamiento de un transistor en drenador abierto con resistencia de pull-up

La figura 11.11 muestra el funcionamiento de dos transistores conectados en drenador abierto. Observando la figura vemos que si ninguno pone un cero, la salida es uno. Pero desde que uno ponga un cero, la salida será cero. Esta configuración también se llama and-cableada (*wired-and*), porque funciona como una puerta *and*.

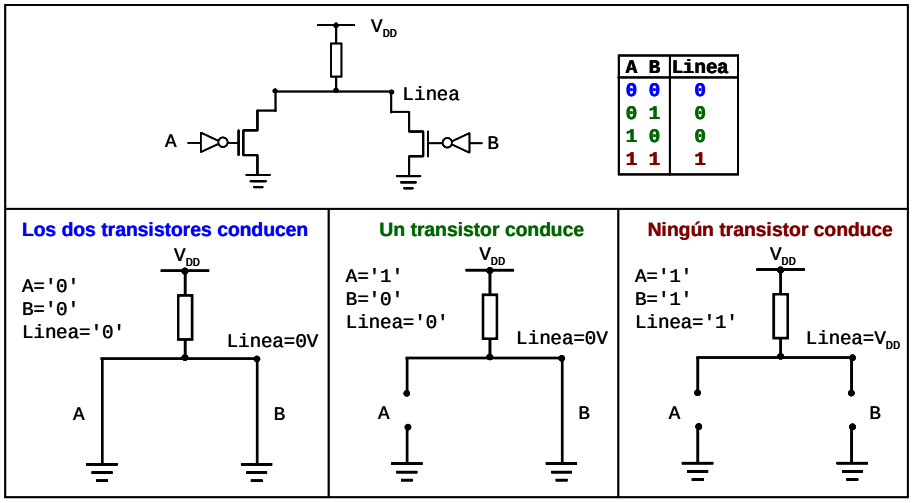


Figura 11.11: Funcionamiento de dos transistores conectados en drenador abierto con resistencia de pull-up

Se pueden conectar más de dos señales en drenador abierto y el funcionamiento sería el mismo, como si fuese una puerta *and* de más de dos entradas.

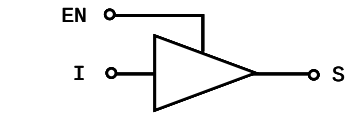
De esta manera es como funcionan las líneas PS/2. En reposo, cuando nadie pone ningún valor, la línea está a uno por la resistencia de *pull-up*. Cuando uno de los dos pone un cero, la línea se pone a cero.

Ahora queda ver cómo se describen estos puertos en VHDL

11.2.4. Puertos bidireccionales en VHDL

Para describir un puerto bidireccional en VHDL se utilizan puertos de tipo `inout` y señales tri-estado. Una señal tri-estado puede valer `'0'`, `'1'` ó puede estar en alta impedancia. En VHDL el valor de alta impedancia se representa como `'Z'`.

Una señal tri-estado se implementa con un *buffer* tri-estado (figura 11.12). Este *buffer* tiene una señal de habilitación (`en`) que cuando está a cero la salida está deshabilitada, o lo que es lo mismo, estará en alta impedancia.



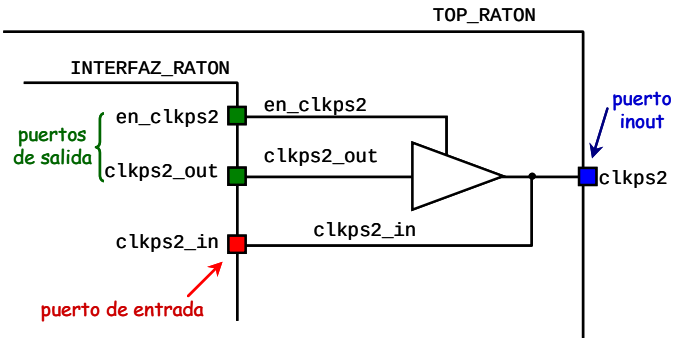
EN	I	S	
0	0	Z	deshabilitada
0	1	Z	
1	0	0	
1	1	1	

```
S <= I when EN = '1' else 'Z';
```

Figura 11.12: Buffer tri-estado

La descripción de un buffer tri-estado en VHDL también se muestra en la figura 11.12.

En VHDL, para los puertos de entrada y salida (`inout`) se recomienda separar la entrada de la salida en la unidad de más alto nivel de modo que no estemos manejando señales de entrada y salida en todo el diseño. La figura 11.13 muestra un ejemplo para un diseño estructural, en donde el puerto de entrada y salida `clkps2` se separa en la señal de entrada `clkps2_in` y la de salida `clkps2_out`. Esta última, habilitada por la señal `en_clkps2`.




```
entity TOP_RATON is
 port (
  clkps2   : inout std_logic;
  ...
end TOP_RATON;

architecture ESTRUCTURAL of TOP_RATON is
 signal en_clkps2 : std_logic;
 signal clkps2_out, clkps2_in: std_logic;
 ...
begin
 clkps2_in <= clkps2;
 clkps2 <= clkps2out when en_clkps2 = '1'
           else 'Z';
 ...
```

Figura 11.13: Separación de un puerto de entrada y salida en señales

Una vez que hemos separado el puerto de entrada y salida `clkps2`, en el resto del diseño no usaremos el puerto `clkps2` sino que usaremos `clkps2_in` para leer y `clkps2_out` para escribir, como si fuesen señales distintas. Esta es la forma a la que estamos acostumbrados a diseñar, la única diferencia es que cuando queremos escribir algo en la señal de salida `clkps2_out` tenemos que poner la señal de habilitación `en_clkps2` a uno.

Con estas indicaciones ya podrías realizar el interfaz con el ratón. Para depurar la interfaz puedes enviar por la UART los datos recibidos del ratón. Una vez que funcione, puedes probar a pintar un cursor en la VGA según el movimiento del ratón.

En este capítulo hemos visto la información necesaria para realizar interfaces básicos con el teclado y el ratón. Para ampliar la información sobre el protocolo PS/2 y los distintos tipos de comandos que tiene, puedes consultar las referencias [4chu, 6ceng, 23qihw].

12. Memorias

El objetivo de este capítulo es aprender a usar los bloques de memoria internos (BRAM) de las FPGAs de *Xilinx*, y en general, a aprender a describir y usar memorias en VHDL.

El capítulo se organizará de la siguiente manera: primero veremos los bloques de memoria internos de las FPGA de *Xilinx*, luego aprenderemos a describir una memoria ROM de modo que podamos mostrar en la VGA una imagen guardada en este tipo de memorias. Por último aprenderemos a describir una memoria RAM, para ello propondremos un circuito que reciba una imagen por la UART, la guarde en la memoria RAM y la muestre por la pantalla.

12.1. Bloques de memoria RAM (BRAM)

La FPGA de la *XUPV2P* tiene 136 bloques de memoria RAM (BRAM) internos, mientras que la *Spartan3e* de la *Nexys2* tiene 20 bloques[84]. Cada bloque tiene 18 Kbits[85] de los cuales 16 Kbits son para guardar datos y 2 Kbits son para bits de paridad. Estos bloques se pueden configurar con distintos anchos de palabra, pudiendo pasar de 16 K posiciones[86] de memoria con un bit de ancho de palabra, hasta 512 posiciones de memoria con palabras de 32 bits (con 4 bits de paridad opcionales). Los bits de paridad se pueden usar como bits adicionales de dato, pero solamente en ciertas configuraciones. En la tabla 12.1 se muestran las distintas configuraciones. Las 3 configuraciones de la derecha (sombreadas) incluyen el bit de paridad, por eso suman 18 Kbits, mientras que las de la izquierda, que no lo usan, suman 16 Kbits.

Posiciones de memoria x número de bits de la palabra					
16K x 1 bit	8K x 2 bits	4K x 4 bits	2K x 9 bits	1K x 18 bits	512 x 36 bits

Tabla 12.1: Configuraciones de las BRAM

Si implementamos una memoria con otro ancho de palabra, perderemos bits. Si queremos una memoria de tres bits de ancho de palabra, usaríamos la configuración 4K x 4 bits, pero perderíamos el cuarto bit (además del bit de paridad que pierde esta configuración).

El tamaño total de la memoria interna de las FPGAs usando las BRAM lo calculamos multiplicando el número de posiciones de memoria de una BRAM por el número de bloques BRAM que hay. Este cálculo se muestra en la tabla 12.2.

	1 bit	2 bits	4 bits	9 bits	18 bits	36 bits
XUPV2P (136)	2176 K	1088 K	544 K	272 K	136 K	68 K
Nexys2 (20)	320 K	160 K	80 K	40 K	20 K	10 K

Tabla 12.2: Tamaño máximo de la memoria usando todas las BRAM de la FPGA según el ancho de palabra

[84] Estos bloques de memoria están dentro de la FPGA, no son las memorias de las placas, por eso son memorias pequeñas.

[85] 1 Kbit = 2^{10} bits = 1024 bits

[86] Estas 16 K posiciones en realidad son 16 x 1024 posiciones.

Así, por ejemplo, con un ancho de palabra de 9 bits, la memoria máxima que podremos usar en la *XUPV2P* tendrá 272 K posiciones (278528), y la *Nexys2* tendrá 40 K posiciones (40960).

Veamos con un ejemplo el tamaño máximo de memoria que disponemos en las FPGAs. Supongamos que queramos guardar una imagen cuadrada en escala de grises con 8 bits de profundidad de color, es decir 256 niveles de gris[87]. Para este tipo de imágenes utilizaríamos un ancho de palabra de 9bits (tabla 12.2). El tamaño máximo de las imágenes según el número de imágenes que queremos guardar se muestra en la tabla 12.3.

8 bits/píxel	1 imagen	2 imágenes	3 imágenes	4 imágenes
XUPV2P	527x527	373x373	304x304	263x263
Nexys2	202x202	143x143	116x116	101x101

Tabla 12.3: Tamaño máximo de las imágenes según el número de imágenes que se van a guardar en las BRAM. Para imágenes de 8 bits por píxel.

Realmente no es mucha cantidad de memoria, si quisiésemos guardar imágenes más grandes tendríamos que recurrir a la memoria externa. La *XUPV2P* puede llevar una DDR SRAM de 256 ó 512 MB; la *Nexys2* tiene una memoria SDRAM y otra *flash* de 16 MB cada una, estas memorias son tres órdenes de magnitud mayores que la memoria interna de la FPGA. Sin embargo, estas memorias externas son más difíciles de usar, especialmente la de la *XUPV2P*, y también son bastante más lentas que las BRAM internas de las FPGAs.

Las memorias BRAM son de doble puerto, esto es, se puede acceder a ellas desde dos puertos independientes, incluso estos puertos pueden funcionar con relojes diferentes. Además, estas memorias son muy rápidas ya que devuelven el dato solicitado en un ciclo de reloj.

Como ejemplo, en la figura 12.1 se muestra el bloque de una BRAM de doble puerto y con relojes diferentes. Esta memoria es la más general, ya que tiene doble puerto de entrada y salida y relojes independientes para cada puerto. No es necesario incluir todos estos puertos al describir una memoria. En este ejemplo la longitud de palabra (`dina`, `douta`, `dinb` y `doutb`) es de 8 bits y el bus de direcciones (`addra` y `addrb`) tiene 17 bits. Con 17 bits se puede acceder a 131072 direcciones[88], así que con esta memoria podríamos guardar una imagen de 362x362 píxeles de 8 bits cada píxel. Según la tabla 12.3, en la *XUPV2P* podríamos tener dos memorias de este tipo, sin embargo, esta memoria no cabría en la *Nexys2*.

[87] Esta imagen la podríamos mostrar completamente con el conversor VGA de la *XUPV2P*, ya que la VGA de esta placa tiene 8 bits para cada color. Sin embargo la *Nexys2* sólo tiene 8 bits para los tres colores (recuerda el apartado 9.2), por lo que estaríamos guardando información que no podríamos mostrar. En el caso de la *Nexys2*, si queremos aprovechar mejor la memoria, podríamos guardar una imagen en color de 8 bits (256 colores). Esto lo veremos en el apartado 12.2.5. En ambos casos, estaríamos guardando una imagen en la que usamos 8 bits para cada píxel.

[88] $2^{17} = 131072$

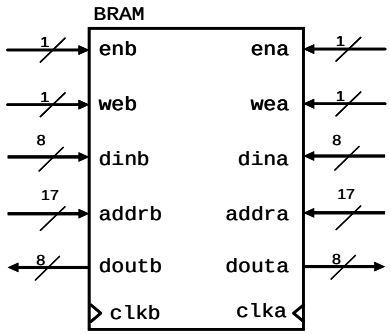


Figura 12.1: Bloque de memoria RAM de doble puerto

En la tabla 12.4 se resume el significado de los puertos.

Señal	bits	I/O	Descripción
`clka`	1	I	Señal de reloj para el puerto A
`ena`	1	I	Habilitación del puerto A, sin habilitación no se podrá leer ni escribir en el puerto A
`wea`	1	I	*Write Enable* de `A`: Habilitación para escritura del dato `dina` en la dirección de memoria `addra`
`dina`	8	I	Dato de entrada para escribir en memoria por el puerto A
`addra`	17	I	Dirección de memoria que se quiere leer/escribir para el puerto A
`douta`	7	O	Lectura del dato que hay en la posición de memoria `addra` (tarda un ciclo de reloj)
`clkb`	1	I	Señal de reloj para el puerto B
`enb`	1	I	Habilitación del puerto B, sin habilitación no se podrá leer ni escribir en el puerto B
`web`	1	I	*Write Enable* de `B`: Habilitación para escritura del dato `dinb` en la dirección de memoria `addrb`
`dinb`	8	I	Dato de entrada para escribir en memoria por el puerto B
`addrb`	17	I	Dirección de memoria que se quiere leer/escribir para el puerto B
`doutb`	7	O	Lectura del dato que hay en la posición de memoria `addrb`

Tabla 12.4: Listado de puertos de la BRAM de doble puerto

12.2. Dibujar una imagen guardada en una ROM

En este apartado vamos a diseñar varios circuitos que dibujen una imagen guardada en distintas memorias ROM. Empezaremos con un diseño muy sencillo y luego lo iremos complicando. Consideraremos las diferencias entre las placas *XUPV2P* y la *Nexys2*. El esquema general de lo que queremos hacer se muestra en la figura 12.2.

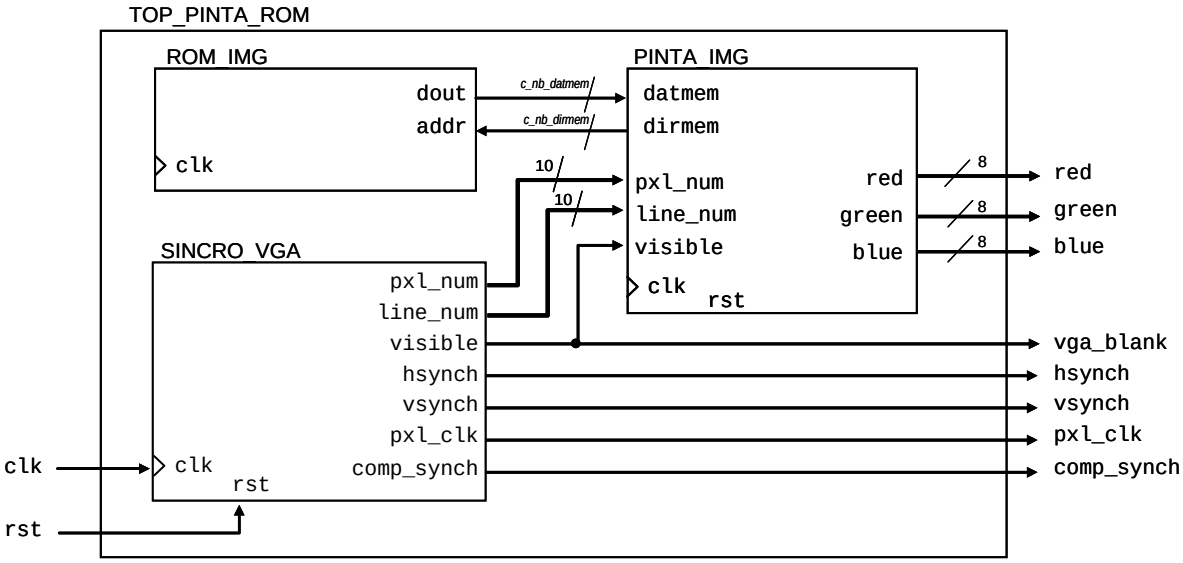


Figura 12.2: Esquema de bloques del circuito que pinta en la pantalla una imagen guardada en una ROM

Podemos ver que el esquema se parece al controlador VGA que vimos en la figura 9.10, la diferencia está en la introducción del bloque de la memoria ROM (`ROM_IMG`).

Observa que el bloque de la memoria ROM es mucho más sencillo que el de la memoria RAM de doble puerto (figura 12.1). Por un lado, esta memoria ROM sólo tiene un puerto, así que con esto se han quitado la mitad de los puertos (`enb`, `web`, `dinb`, `addrb`, `doutb` y `clkb`). Por otro lado, por ser una memoria de sólo lectura, se quita el dato de entrada `dina` y la habilitación de escritura `wea`. Tampoco se ha incluido la señal de habilitación `ena`. Así que sólo nos queda el reloj (`clk`), la dirección de memoria que queremos leer (`addr`) y el dato devuelto por la memoria (`dout`).

Observa que el número de bits de la dirección de memoria (`c_nb_dirmem`) y el número de bits del dato[89] (`c_nb_datmem`) están con constantes. Y también que el número de bits de los colores (`red`, `green` y `blue`) son ocho, pero para la *Nexys2* serán 3, 3 y 2 respectivamente.

Ahora veremos cómo se puede describir una memoria ROM.

12.2.1. Memoria ROM de un bit de ancho de palabra

En VHDL, una memoria ROM es simplemente una constante de tipo vectorial a la que se accede mediante un proceso con reloj. La dirección de memoria de la ROM es el índice del vector. En el código 12.1 se muestra un módulo que describe una memoria ROM de 256 posiciones y con datos de un único bit. Esta memoria ROM representa una imagen de 16 filas y 16 columnas. Por eso se ha llamado `ROM1b_16x16`, sin embargo la memoria es unidimensional, es decir, accedemos por un índice (`addr`) que va desde 0 a 255 y no por la fila y columna. Como son 255 posiciones de memoria `addr` es de 8 bits.

[89] El ancho de palabra

```
entity ROM1b_16x16 is
  port (
    clk  : in  std_logic;
    addr : in  std_logic_vector(7 downto 0);
    dout : out std_logic
  );
end ROM1b_16x16;

architecture BEHAVIORAL of ROM1b_16x16 is
  signal addr_int  : natural range 0 to 2**8-1;
  type memostruct is array (natural range<>) of std_logic;

  constant img : memostruct := (
 -- 0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
   '1','0','1','0','1','0','1','0','1','0','1','0','1','0','1','0', -- 0
   '1','0','1','0','1','0','1','0','1','0','1','0','1','0','1','0', -- 1
   '1','0','1','0','1','0','1','0','1','0','1','0','1','0','1','0', -- 2
   '1','0','1','0','1','0','1','0','1','0','1','0','1','0','1','0', -- 3
   '1','0','1','0','1','0','1','0','1','0','1','0','1','0','1','0', -- 4
   '1','0','1','0','1','0','1','0','1','0','1','0','1','0','1','0', -- 5
   '1','0','1','0','1','0','1','0','1','0','1','0','1','0','1','0', -- 6
   '1','0','1','0','1','0','1','0','1','0','1','0','1','0','1','0', -- 7
   '1','1','1','1','1','1','1','1','1','1','1','1','1','1','1','1', -- 8
   '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0', -- 9
   '1','1','1','1','1','1','1','1','1','1','1','1','1','1','1','1', --10
   '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0', --11
   '1','1','1','1','1','1','1','1','1','1','1','1','1','1','1','1', --12
   '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0', --13
   '1','1','1','1','1','1','1','1','1','1','1','1','1','1','1','1', --14
   '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'  --15
  );
begin

  addr_int <= TO_INTEGER(unsigned(addr));  -- pasar a entero la direccion

  P_ROM: process (clk)
  begin
    if clk'event and clk='1' then
      dout <= img(addr_int);
    end if;
  end process;

end BEHAVIORAL;
```

Código 12.1: Memoria ROM de 256 posiciones y un bit de ancho de palabra[90]

En el código 12.1 podemos ver que el dato de salida `dout` se obtiene un ciclo de reloj después de haber introducido la dirección que queremos. Esto tiene implicaciones porque retrasará un ciclo de reloj las salidas que dependen de este dato.

12.2.2. Circuito que dibuja la imagen de la ROM de un bit de ancho

Queremos dibujar la imagen de 16x16 de la ROM (código 12.1) en la esquina superior izquierda de la VGA. El resto de la pantalla estará en color azul.

La imagen de la ROM sólo tiene un bit por píxel, por tanto, será una imagen de dos colores: blanco y negro[91]. Si consideramos los unos como blanco y los ceros como negro, la imagen sería como la mostrada en la figura 12.3. En la imagen se han pintado los bordes de cada píxel con línea discontínua gris para apreciar más fácilmente sus dimensiones.

[90] El código de esta memoria se puede descargar de la referencia [28web]

[91] En realidad podemos elegir otros dos colores cualquiera, pero sólo dos.

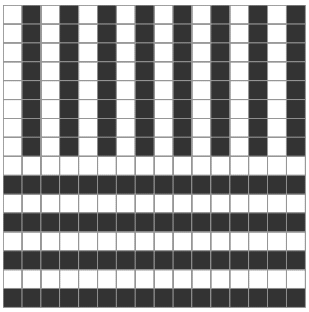


Figura 12.3: Imagen de la ROM del código 12.1

Dibujaremos la imagen en la esquina superior izquierda de la pantalla. Como es una imagen de 16x16, estará dibujada entre las columnas 0 y 15, y entre las filas 0 y 15 de la pantalla. Esta será la condición que tendremos que poner en el proceso del módulo `PINTA_IMG` (figura 12.2), que es el encargado de pintar[92]. El resto de la pantalla la pintaremos de azul.

Para mostrar la imagen tendremos que pedir el dato de la memoria, esperar a que la memoria devuelva el dato y dibujar el píxel en la VGA. Todo este proceso se tiene que hacer rápido, ya que la VGA no espera. La VGA tiene que mostrar los píxeles consecutivamente en los tiempos determinados por su frecuencia de refresco (recuerda la tabla 9.1). Por suerte las memorias BRAM son rápidas y sólo tardan un ciclo de reloj en devolver el dato solicitado. En nuestra VGA cada píxel se muestra a una frecuencia de 25 MHz, y los relojes de las placas van cuatro veces más rápido en la *XUPV2P* o dos veces más rápido en la *Nexys2*. Así que en principio no debería de haber problema por retrasar un ciclo de reloj.

En la figura 12.4 se muestra cómo realizar el cálculo de la dirección de memoria (`dirmem` o `addr`) a partir del número de píxel (`pxl_num`) y el número de línea (`line_num`) de la VGA.

La figura representa la esquina superior izquierda de la VGA, donde queremos dibujar la imagen de 16x16. Los números en azul representan las columnas o píxeles de la VGA, que aunque van de 0 a 639, en la figura sólo se han dibujado los primeros 18 (de 0 a 17). Los números rojos representan las líneas (filas) de la VGA, que sólo se han dibujado de la línea cero a la cuatro. Los cuadrados representan los píxeles de la imagen, de todos los píxeles de la imagen sólo se han dibujado los de las primeras filas, aunque en éstas sí se muestran las 16 columnas de cada fila. Por eso la imagen no se ve cuadrada, aunque lo sea. Dentro de cada píxel se ha indicado el número de píxel de la imagen. Este número de píxel se corresponde con la dirección de memoria que hay que solicitar para obtener el color del píxel.

[92] sería el proceso equivalente al del código 9.2, pero para este diseño el proceso será más pequeño

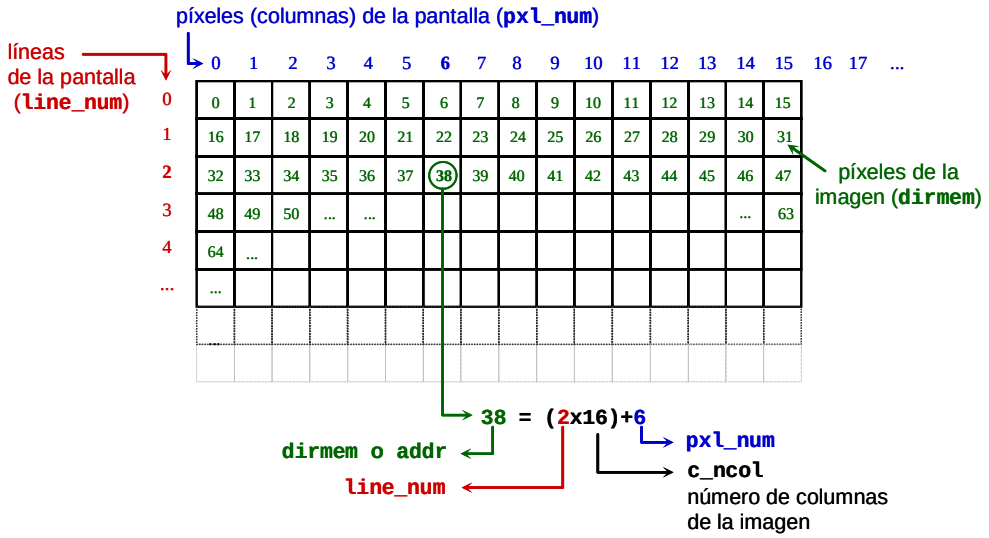


Figura 12.4: Cálculo de la dirección de memoria a través del número de píxel y número de línea de la VGA

En la parte inferior de la figura 12.4 se muestra cómo calcular la dirección de memoria correspondiente al píxel que tenemos que mostrar en cada momento. La formula es:

$$\text{dirmem} = (\text{line\_num} \cdot \text{c\_ncol}) + \text{pxl\_num} \qquad (12.1)$$

Donde `dirmem` es la dirección de memoria (también la hemos llamado `addr`); `pxl_num` y `line_num` son el número de píxel (columna) y el número de línea de la VGA, generados en el sincronizador de la VGA (`SINCRO_VGA`); y `c_ncol` es una constante que define el número de columnas de la imagen que estamos dibujando, que en el ejemplo vale 16.

Recuerda que para realizar una multiplicación las dimensiones de los factores y el producto son muy importantes, si no, se sintetizarán multiplicadores muy grandes que consumirán muchos recursos. Así que tenemos que analizar las operaciones implicadas, y especialmente la multiplicación:

- El producto no podrá ser mayor que 255 (8 bits), pues es el número de direcciones de la memoria, y no podemos acceder a una posición de memoria mayor.
- Aunque `line_num` tiene un rango de 0 a 519 (10 bits), en realidad, para nuestra imagen el máximo número de filas son 15 (4 bits). Por lo tanto, en vez de usar `line_num` en su rango completo, usaremos `line_num(3 downto 0)`.
- Algo similar ocurre con `pxl_num`, aunque su rango es de 0 a 799, para nuestra imagen sólo necesitamos los cuatro primeros bits: `pxl_num(3 downto 0)`.
- El resultado de multiplicar el valor máximo de `line_num(3 downto 0)` por 16, es un número de 8 bits (15x16=240).
- Sumando el número anterior con el mayor número que puede tener `pxl_num(3 downto 0)` resulta en 255=240+15.

Así que la fórmula 12.1 quedará[93]:

$$\text{dirmem} = (\text{line\_num}(3{:}0) \cdot \text{c\_ncol}) + \text{pxl\_num}(3{:}0) \qquad (12.2)$$

Sin embargo, si pasamos esta multiplicación directamente a VHDL no se sintetizará como queremos. Recuerda los códigos 8.20 y 8.21, en los que veíamos que no podemos multiplicar una señal por una constante de mayor rango. En nuestro caso, `n_col` tiene 5 bits mientras que `line_num(3:0)` tiene cuatro bits. En VHDL tendríamos que hacerlo como

[93] Esto es una fórmula, no tiene la sintaxis del VHDL

muestra el código 12.2. Se ha tenido que crear la señal intermedia `line_numx16`, porque el producto tendrá diez bits. Tiene que tener diez bits porque el factor tiene cinco bits y al multiplicar por una constante el producto tendrá el doble de bits que el factor no constante. Posteriormente, en la suma se trunca `line_numx16` ya que hemos visto que en nuestro caso el producto nunca va a tener más de ocho bits.

```
...
  signal   line_numx16 : unsigned (9 downto 0);
begin
  line_numx16 <= ('0'&line_num(3 downto 0)) * 16;
  dirmem      <= line_numx16(7 downto 0) + pxl_num(3 downto 0);
...
```

Código 12.2: Cálculo de la dirección de memoria con multiplicación

Haciendo la multiplicación de esta manera evitamos que se sinteticen multiplicadores más grandes de lo necesario y obtenemos un rango acorde con la dirección de memoria.

El cálculo de la dirección de memoria no hace falta que se haga únicamente cuando estemos en el recuadro de la VGA que corresponde con la imagen, es decir entre las filas y columnas 0 y 15. Este módulo puede estar constantemente calculando la dirección de memoria y solicitando a la memoria datos, pero los datos devueltos sólo los utilizaremos cuando estemos pintando la imagen (esquina superior izquierda). Esta es una de las diferencias entre el software y el hardware. El circuito encargado de calcular la dirección de memoria es independiente del resto y el que esté calculando la dirección constantemente no hace que el resto del circuito vaya más lento[94].

Con estas indicaciones puedes realizar el circuito. Cuando la pantalla muestre la imagen, debes de comprobar que se dibuja una imagen cuadrada de 16x16 como la de la figura 12.3. Dieciséis píxeles de ancho no son muchos para una pantalla y por eso la imagen es pequeña, pero queríamos evitar incluir una ROM muy grande en el código 12.1. Debes de comprobar que las líneas son rectas y continuas; que tienen el mismo ancho, especialmente en los bordes y esquinas; y que la imagen es cuadrada.

Si no se mostrase correctamente, deberás de simular y ver los valores de los puertos `red`, `green` y `blue` en los píxeles y líneas de la VGA donde se pinta la imagen. Haz las comprobaciones una vez que haya pasado la primera sincronización vertical (`hsynch='0'`). Leer el siguiente apartado también te puede ayudar a encontrar algún fallo.

12.2.3. Mejoras y variantes del circuito que dibuja la imagen de la ROM

En este apartado veremos algunas mejoras y variantes del circuito anterior. Estas variantes podrán ser útiles para entender los siguientes apartados y capítulos.

Haz estas modificaciones en otras carpetas y proyectos, copiando los ficheros VHDL. Es muy importante que cuando realices modificaciones de un circuito que funciona mantengas la versión original. Así, en caso de que nada funcione siempre puedas volver atrás.

12.2.3.1. Retraso de las salidas de la VGA

Hemos visto que al solicitar los datos a la memoria vamos a tener un retraso de un ciclo de reloj en las señales que se generan en el módulo `PINTA_IMG`: `red`, `green` y `blue`. Sin embargo, el resto de señales de la VGA no tienen retraso. Esto no es un gran problema

[94] Sin embargo, en el caso de que queramos que nuestro circuito consuma poca energía es conveniente parar los circuitos que no estén haciendo nada útil.

porque la frecuencia de la FPGA es mayor que la de la VGA. Sin embargo, a veces da problemas, especialmente en los primeros píxeles de cada fila[95]. Debido a esto, puede ser que las primeras columnas no se muestren bien.

Como solución se puede retrasar el resto de señales de la VGA haciendo que estén sincronizadas con las señales que se generan en `PINTA_IMG`. Realizando esta modificación, el esquema de la figura 12.2 se transformaría en el de la figura 12.5, en donde el único cambio es la inclusión del módulo `VGA_REG` para registrar las señales de la VGA. Al registrar las señales (código 2.13), en el reset se debería de asignar el valor que haga inactiva la señal[96].

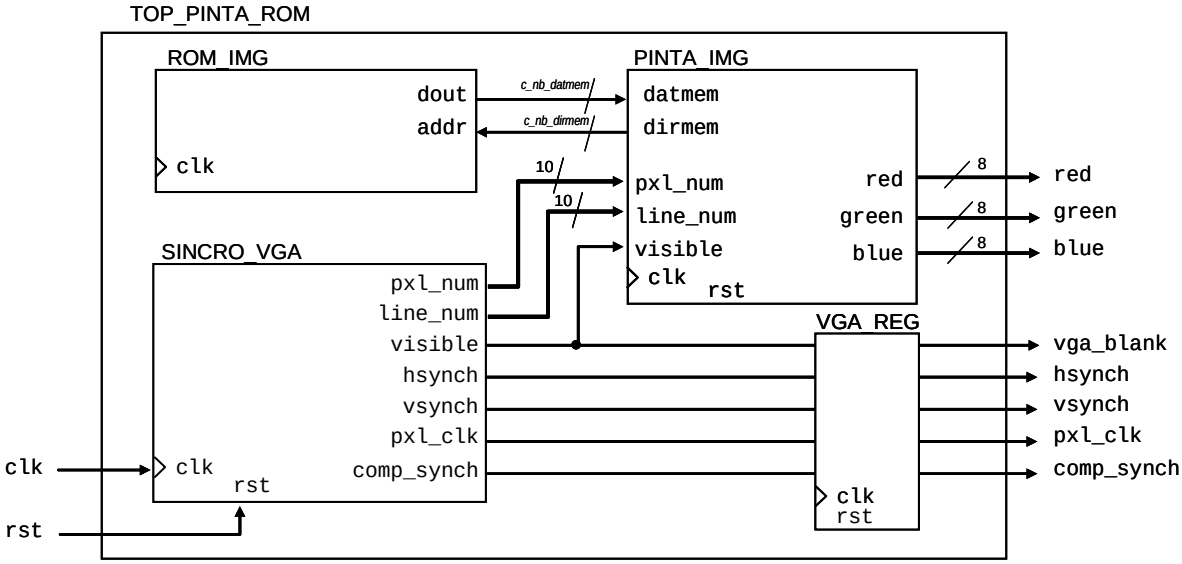


Figura 12.5: Esquema de bloques del circuito que pinta en la pantalla una imagen guardada en una ROM retrasando las señales de la VGA

Tanto si se produce algún defecto de este tipo como si no, te recomendamos que cambies el circuito y registres las salidas. Esto, además de sincronizar las señales, registra los puertos de salida. Registrar los puertos de salida es una práctica muy recomendada porque la señal no tendrá los retardos ni las transiciones espurias debidas a la lógica combinacional.

Las transiciones espurias se producen por los retardos de las puertas (y también las interconexiones). En la figura 12.6 se puede ver cómo al cambiar la entrada `A` y `D`, se produce una transición espuria en la salida `S3`. La transición es espuria porque después de los retardos debidos a la propagación de las señales, la salida `S3` mantendrá su valor original, por lo tanto son transiciones no deseadas. Si registramos esta salida, la transición espuria no se trasladará a la salida del biestable (`P0`) porque el tiempo de esta transición es menor que el periodo del reloj.

[95] En nuestro diseño la segunda columna (la negra) de la imagen (figura 12.3) se veía blanca, por lo que se unía con la primera y tercera columna, pareciendo una columna más gruesa que el resto. Esto se producía en la *XUPV2P* pero no en la *Nesys2*

[96] Por ejemplo, el valor inactivo de las señales `hsynch` y `vsynch`, suele ser '1' y lo definimos por la constante `c_synch_act` invertida (recuerda el código 9.1).

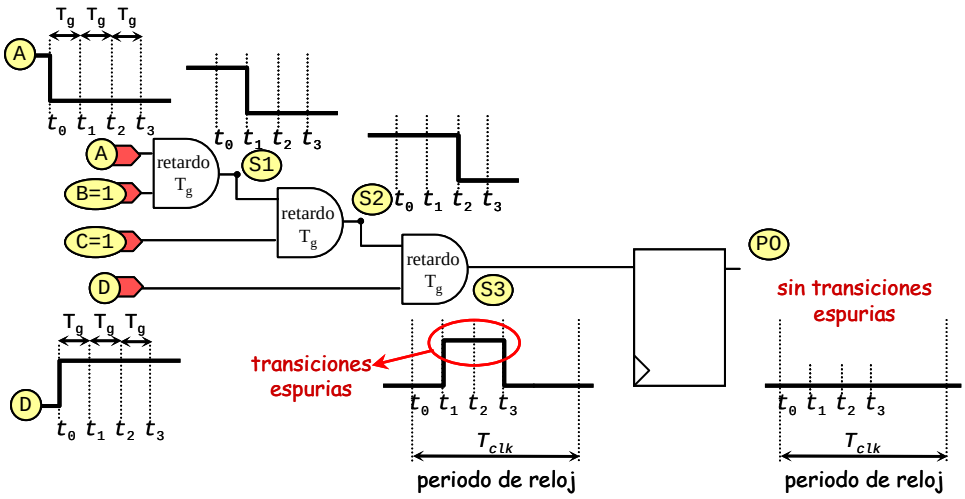


Figura 12.6: Eliminación de transiciones espurias con un biestable

Si implementas el circuito propuesto en este apartado (figura 12.5) y te sale bien, mantenlo como circuito base para los demás cambios. Es decir, parte de este para hacer los cambios propuestos en los siguientes apartados.

12.2.3.2. Cálculo de las direcciones sin multiplicador

En la fórmula del cálculo de la dirección de memoria vemos que hay una multiplicación. Del capítulo 8, sabemos que las multiplicaciones consumen bastantes recursos y son más lentas. Una alternativa a la multiplicación es utilizar un registro que se inicialice en la fila cero y que aumente en 16 (`c_ncol`) cada vez que se incremente una fila. Hacerlo de esta manera requiere un poco más de cuidado en las sumas y las inicializaciones de este registro, pero ahorra un multiplicador. Si quieres, puedes hacer una primera versión con la multiplicación y luego probar a realizarlo de esta manera.

12.2.3.3. Dibujar la imagen en otro lugar de la pantalla

En vez de dibujar la imagen en la esquina superior izquierda, empezando por el píxel de la posición (0,0), ahora realizaremos el mismo circuito pero dibujando la imagen a partir de la fila 100 y columna 200.

La solución es fácil, en el proceso de pintar hay que cambiar las condiciones que sitúan dónde se pintan los píxeles de la imagen (proceso `P_pinta`). Por otro lado, hay que cambiar la ecuación del cálculo de las direcciones. Ahora ya no se calculan con `pxl_num` y `line_num`, sino que hay que hacer un cambio de ejes restando 200 y 100 a estas señales respectivamente.

Puedes pensar qué pasará cuando al realizar estas restas `pxl_num` y `line_num` sean menores que los sustraendos (200 y 100). En realidad no nos importa porque en el proceso `P_pinta` se controla que sólo se pinte cuando sean mayores. Pero no está mal que pienses qué pasaría si no se controlase.

12.2.3.4. Repetición de la imagen

Hemos realizado el circuito de modo que el cálculo de la dirección de memoria se realiza constantemente y en el proceso de pintar (proceso `P_pinta`) es donde se incluyen las condiciones que indican dónde se pinta la imagen. En esta propuesta vamos a quitar las condiciones en el proceso `P_pinta` que hacen que la imagen se dibuje sólo en la esquina superior izquierda. Así que el proceso de pintar quedaría como muestra en el código 12.3.

```
  P_pinta: Process (visible, datmem)
  begin
    red   <= (others=>'0');
    green <= (others=>'0');
    blue  <= (others=>'0');
    if visible = '1' then
      red   <= (others => datmem);
      green <= (others => datmem);
      blue  <= (others => datmem);
    end if;
  end process;
```

Código 12.3: Dibujar la imagen de manera repetida

El resultado será que la imagen se dibuja de manera repetitiva por toda la pantalla. Puedes probar a cambiar a mano la imagen de la ROM del código 12.1 para poner otras formas.

¿Entiendes por qué se forma esa imagen en la VGA?

12.2.3.5. Ampliación por dos

En esta variante vamos a volver a pintar una sola imagen en la esquina superior izquierda, pero la queremos pintar al doble de tamaño. En vez de dibujar la imagen en su tamaño original de 16x16, la pintaremos a tamaño 32x32. ¿Se te ocurre cómo hacerlo?

Por un lado tienes que cambiar el proceso `P_pinta` de modo que ahora el cuadrado donde se pinta la imagen sea de 32x32 (esquina superior izquierda de la pantalla). Por otro lado queremos pintar el mismo píxel de la imagen en la intersección de dos columnas y dos filas de la pantalla. Para ello, al calcular la dirección de memoria tendremos que dividir entre dos los números de columna y fila en la que estamos.

La figura 12.7 muestra cómo se calcularían las direcciones de memoria. A diferencia de la figura 12.4, para evitar una imagen muy grande, en esta figura no se muestran las 32 columnas de la imagen.

Los cuadros con borde verde representan cada píxel de la imagen. Estos cuadros abarcan cuatro píxeles de la pantalla (borde negro). Observa que quitando el bit menos significativo de `pxl_num` obtenemos la columna de la imagen en la que estamos. Y lo mismo sucede con la fila, por ejemplo, los píxeles de la fila cero de la imagen están en las filas cero y uno de la pantalla. Como ya sabemos, quitar el bit menos significativo es dividir por dos.

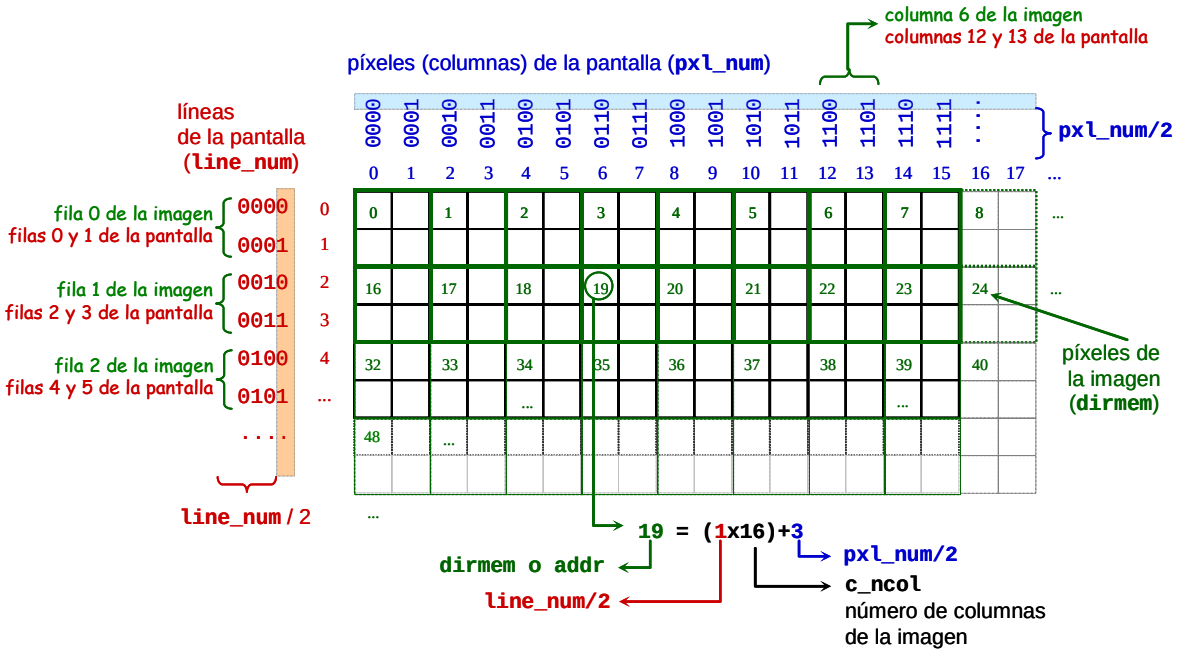


Figura 12.7: Cálculo de la dirección de memoria para pintar una imagen el doble de grande

Ahora, en vez de realizar las operaciones con `pxl_num(3 downto 0)` y `line_num(3 downto 0)` como se hicieron en el código 12.2, las haremos con `pxl_num(4 downto 1)` y `line_num(4 downto 1)`. Con esto convertimos la intersección de dos filas y dos columnas de la pantalla en un sólo píxel de la imagen, dibujando la imagen con tamaño 32x32.

Cuando hayas dibujado la imagen al doble de tamaño puedes probar a realizar la ampliación por cuatro, ocho,... Ampliaciones que no sean potencias de dos no son tan fáciles, aunque no tan difíciles como las ampliaciones de numeros fraccionarios.

12.2.3.6. Uso de las potencias de dos

Quizá ya te hayas dado cuenta que para la imagen que estamos utilizando hay una forma más fácil de calcular las direcciones de memoria. Como la imagen es de 16x16, y 16 es una potencia de 2, las multiplicaciones son mucho más sencillas. Lo que hemos hecho anteriormente es válido para cualquier tipo de imagen, pero para imágenes con un un número de columnas que sea potencia de dos hay maneras más sencillas de realizar los cálculos.

Multiplicar un número por una potencia de dos (2^n) es equivalente a desplazar el número, n veces a la izquierda. Como el número de columnas de la imagen es 16 (2^4), hay que desplazar cuatro veces a la izquierda el número de línea de la pantalla (equivalente a la multiplicación por 16 del código 12.2) y sumarle los cuatro bits menos significativos del número de píxel (columna) de la pantalla. Como del número de píxel cogíamos los cuatro bits menos significativos y ahora desplazamos el número de línea de la pantalla cuatro bits a la izquierda, al final no es necesario sumar sino que basta con concatenar, pues las líneas ocupan los 4 bits más significativos y las columnas (píxeles de la pantalla) los 4 bits menos significativos. La figura 12.8 muestra de manera esquemática la operación que hay que hacer.

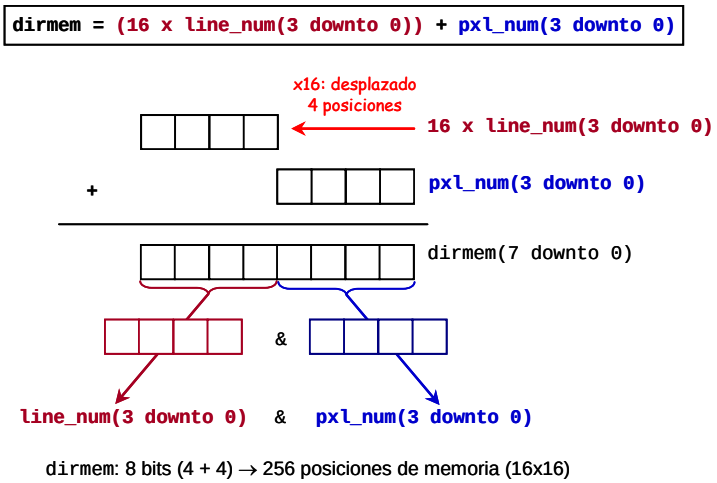


Figura 12.8: Esquema del cálculo de la dirección de memoria para imágenes de 16x16

Por lo tanto, en este caso el cálculo de la dirección de memoria se reduce a concatenar los cuatro bits menos significativos de las líneas y los píxeles de la pantalla.

Así que en estos casos el cálculo de la dirección de memoria resulta muy fácil. Para ayudar a entender mejor por qué es tan fácil, en la figura 12.9 se ha modificado la figura 12.4 poniendo los números de líneas y píxeles en hexadecimal. Haciéndolo así, se puede ver que la obtención del número de píxel de la imagen (dirección de memoria) se consigue concatenando el número de línea con el número de píxel de la pantalla.

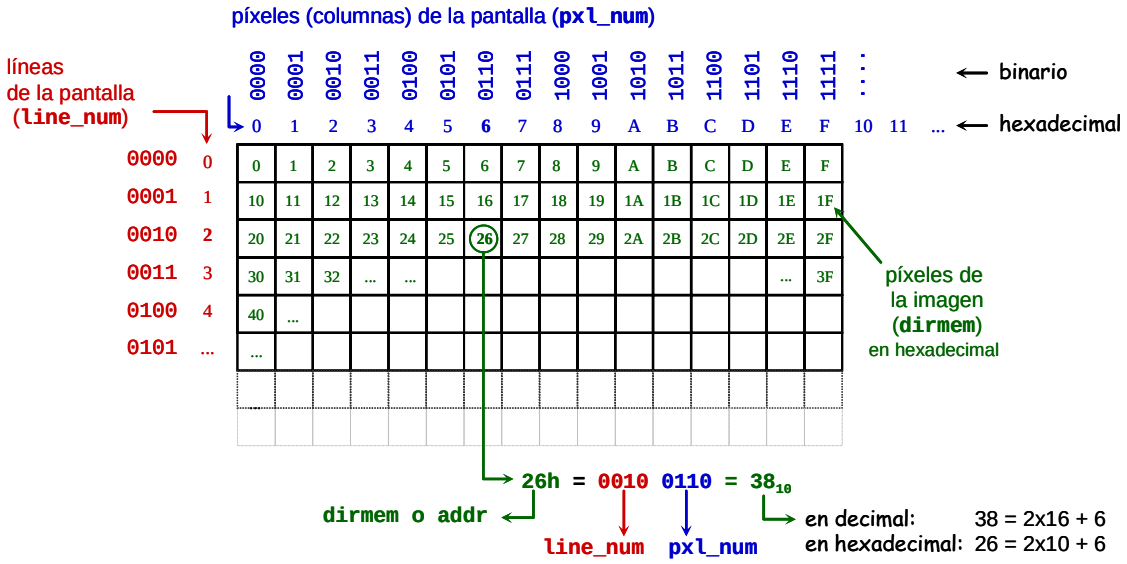


Figura 12.9: Cálculo de la dirección de memoria aprovechando que la imagen tiene un número de columnas potencia de dos

Haz un nuevo proyecto incorporando esta forma de calcular la dirección de memoria. Emplearemos mucho esta forma de cálculo en el capítulo 15 en el que diseñaremos el videojuego *Pac-man*.

12.2.4. Memoria ROM de varios bits de ancho de palabra

Cuando cada píxel de la imagen es de un único bit[97] se pueden agrupar todas las columnas de una línea en una única dirección de memoria. Por ejemplo, en vez de definir

[97] Es una imagen en blanco y negro

la memoria como lo hicimos en el código 12.1, podemos describir la ROM como lo hace el código 12.4, en donde todos los bits/píxeles de una fila están en un vector que se guarda en una única posición de memoria.

```
entity ROM16b_16 is
  port (
    clk  : in  std_logic;   -- reloj
    addr : in  std_logic_vector(3 downto 0);
    dout : out std_logic_vector(15 downto 0)
  );
end ROM16b_16;

architecture BEHAVIORAL of ROM16b_16 is
  type memostruct is array (natural range<>) of std_logic_vector(15 downto 0);

  constant img : memostruct := (
   -- FEDCBA9876543210
     "0111111111111110",-- 0
     "1111111111111111",-- 1
     "1111000000001111",-- 2
     "1110000000001111",-- 3
     "1110001111111111",-- 4
     "1110011111111111",-- 5
     "1110011111111111",-- 6
     "1110011111111111",-- 7
     "1110011111111111",-- 8
     "1110011111111111",-- 9
     "1110011111111111",-- A
     "1110001111111111",-- B
     "1110000000001111",-- C
     "1111000000001111",-- D
     "1111111111111111",-- E
     "0111111111111110" -- F
    );
begin

  P_ROM: process (clk)
  begin
    if clk'event and clk='1' then
      dout <= img(to_integer(unsigned(addr)));
    end if;
  end process;

end BEHAVIORAL;
```

Código 12.4: Memoria ROM de 16 posiciones y 16 bits de ancho de palabra[98]

Posiblemente habrás notado que la imagen de la memoria es distinta a la que hemos estado usando, esto lo hemos hecho para incluir algo de variedad e mostrar nuevas posibilidades. Para distinguir mejor la imagen en el propio código se han puesto en negrita los ceros.

Observa cómo han cambiado los anchos de los puertos, ahora la memoria sólo tiene 16 posiciones (las 16 filas de la imagen) y por tanto bastan cuatro bits para acceder a cualquier posición. Por otro lado, el dato tiene 16 bits (las 16 columnas de la imagen).

Guardar todos los píxeles de una fila en una misma dirección de memoria puede tener ventajas:

- Normalmente no hay memorias con un sólo bit de ancho de palabra. Si usamos un sólo bit en una memoria de mayor ancho de palabra, estaríamos desperdiciando el resto de los bits.
- Las memorias tienen tiempos de acceso relativamente largos, si estos tiempos fuesen críticos, obtener toda una fila con un único acceso a memoria podría hacer aumentar la velocidad del sistema.

[98] El código de esta memoria se puede descargar de la referencia [28web]

En nuestro caso no hay gran diferencia entre estas configuraciones de la ROM, ya que las BRAM son muy rápidas y se pueden configurar de cualquiera de las maneras.

¿Cómo calcularíamos ahora la dirección de memoria para obtener cada píxel?

La dirección de memoria es muy fácil de calcular ya que las filas de la imagen se corresponden con las filas de la pantalla[99], por tanto bastaría con asignar a `dirmem/addr` los cuatro bits menos significativos de `line_num`.

Para obtener el píxel a partir del dato devuelto por la memoria, tendremos que seleccionar el bit correspondiente a la columna de la pantalla en la que estamos. Sin embargo, como los índices de la memoria son descendentes[100] y las columnas de la pantalla aumentan de izquierda a derecha, tendremos que realizar una de las siguientes opciones:

- Invertir el orden de los bits (apartado 8.8.3.1)
- Realizar una resta del índice para no ver la imagen simétrica de la imagen original
- Guardar la imagen simétrica en la memoria.

En la figura 12.10 se muestra cómo realizar la resta del índice para obtener el índice complementario al ancho de la imagen.

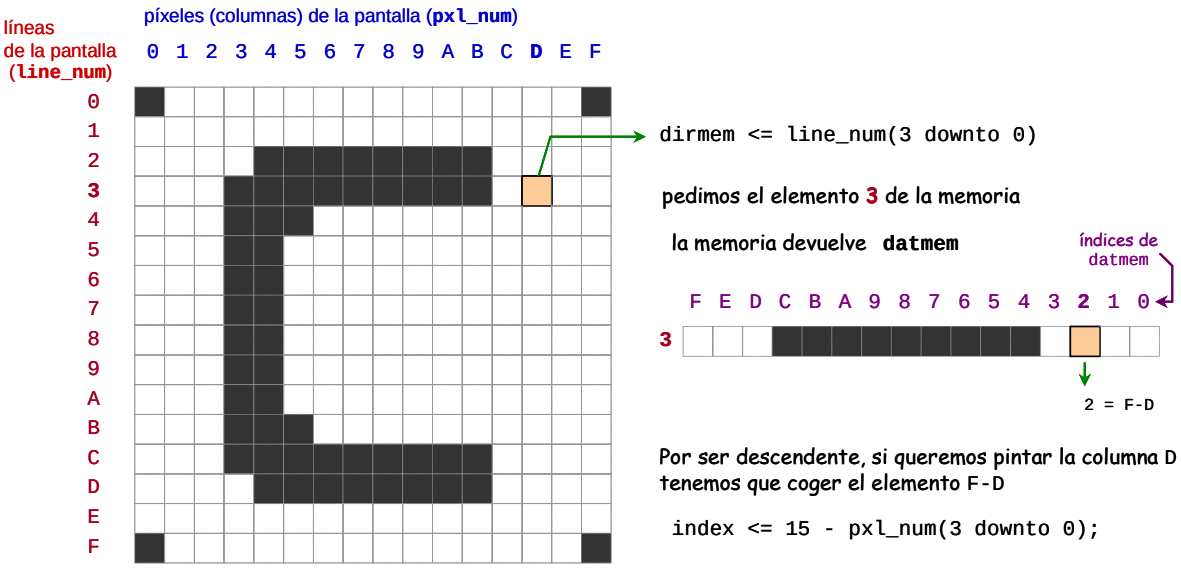


Figura 12.10: Cálculo de la dirección de memoria y del índice del píxel en memorias que guardan toda la fila en una dirección

Realiza el diseño para mostrar esta imagen y comprueba que obtienes la misma imagen que la dibujada en la figura 12.10, verifica que no sale la imagen simétrica y que los cuatro puntos negros de las esquinas se dibujan correctamente.

Dibujar esta imagen nos puede abrir las posibilidades para dibujar caracteres de una manera más eficiente que llenando los procesos VHDL con condiciones *if* y *elsif*. Por ejemplo, si quisiésemos incluir un marcador de goles en el videojuego *Pong* del capítulo 10, sería más eficiente utilizar memorias ROM con los mapas de bits de los caracteres. Esto lo veremos en el capítulo 13, pero está bien que vayas pensando cómo lo harías.

[99] En caso de que estemos pintando la imagen en la esquina superior izquierda, si no, tendremos que hacer una resta para trasladar los ejes de coordenadas. Como hicimos en el apartado 12.2.3.3

[100] En este caso `15 downto 0`

12.2.5. Memoria ROM para imagen con mayor profundidad de color

En este ejercicio pintaremos imágenes en escala de grises o en color. La imagen se guardará en una memoria similar a la del código 12.4 con la diferencia de que ahora cada dirección de memoria guarda un píxel y no una fila de la imagen. La memoria tendrá un ancho de palabra de más de un bit en la que se guardan los distintos tonos de gris o los distintos colores.

Por ejemplo el código 12.5 describe una memoria ROM de una imagen de 10x10 píxeles en escala de grises. Por tanto tendrá 100 posiciones de memoria y en cada posición se guarda un píxel. Esto proporciona ocho bits de información sobre su color, lo que da 256 niveles de gris.

```
entity ROM8b_pacman10x10 is
  port (
    clk  : in  std_logic;
    addr : in  std_logic_vector(7-1 downto 0);
    dout : out std_logic_vector(8-1 downto 0)
  );
end ROM8b_pacman10x10;

architecture BEHAVIORAL of ROM8b_pacman10x10 is
  type memostruct is array (natural range<>) of std_logic_vector(8-1 downto 0);
  constant filaimg : memostruct := (
   "00000111", "00001000", "01001011", "10011110", "11011011", "11100111", "11000010",
   "01111111", "00011101", "00001001", "00001000", "01111101", "11010100", "11101010",
   "11011110", "11001111", "11100110", "11101000", "10111010", "00111101", "01000110",
   "11010110", "11101011", "11101000", "11100111", "11100100", "11101011", "11100100",
   "11110010", "10111001", "10100111", "11101011", "11100011", "11010011", "11100111",
   "11101001", "11100100", "11000000", "01010010", "00010110", "11011111", "11100010",
   "11100011", "11100011", "11100101", "11000010", "01101010", "00100010", "00011110",
   "00000011", "11100011", "11010111", "11100000", "11100011", "11100111", "11000111",
   "01011010", "00001101", "00000101", "00000100", "10101010", "11101001", "11101000",
   "11011100", "11100101", "11100111", "11100000", "11011100", "01110111", "00011001",
   "01001000", "11100011", "11101000", "11010111", "11100110", "11100101", "11100111",
   "11011110", "11011111", "10111001", "00001000", "01100111", "11011000", "11101010",
   "11010110", "11011001", "11101001", "11011101", "10111100", "00111001", "00000101",
   "00010100", "01001001", "10011110", "11010010", "11100101", "11001101", "01110110",
   "00110011", "00000111");
begin
  P_ROM: process (clk)
  begin
    if clk'event and clk='1' then
      dout <= filaimg(to_integer(unsigned(addr)));
    end if;
  end process;
end BEHAVIORAL;
```

Código 12.5: Memoria ROM de 10x10 y 256 colores grises[101]

Esta memoria es similar a la del código 12.1, la diferencia es que en el código 12.1 cada píxel tenía un bit (blanco o negro) y ahora se guarda mucha más información de la intensidad del gris de cada píxel.

El cálculo de la dirección de memoria para mostrar el píxel en la VGA se realiza de manera similar al de la memoria del código 12.1. Aunque ahora, por ser una memoria de 10x10, no se puede calcular concatenando fila y columna de la VGA como se proponía en el apartado 12.2.3.6.

Como la memoria guarda ocho niveles de gris, en la *XUPV2P* la asignación del dato devuelto por la memoria a los puertos de los colores es directa (código 12.6), mientras que en la *Nexys2* hay que truncar porque su conversor VGA tiene menos niveles de color (código 12.6).

[101] El código de esta memoria se puede descargar de la referencia [28web]

```
...
  red   <= datmem;
  green <= datmem;
  blue  <= datmem;
...
```

Código 12.6: Asignación del valor del dato de la memoria de 8 bits de gris a los colores de la XUP

```
...
  red   <= datmem (7 downto 5);
  green <= datmem (7 downto 5);
  blue  <= datmem (7 downto 6);
...
```

Código 12.7: Asignación del valor del dato de la memoria de 8 bits de gris a los colores de la Nexys2

Como alternativa, si queremos mostrar la imagen en color usando la misma cantidad de memoria podríamos guardar en los ocho bits los tres colores RGB. Esto es especialmente conveniente para la *Nexys2* ya que la memoria que acabamos de describir guarda más bits de color que los que se pueden mostrar por la VGA de la *Nexys2*.

El código 12.8 muestra una ROM que es igual que la anterior (código 12.5) pero con la diferencia de que ahora cada elemento de memoria no guarda el color en escala de grises sino representa su combinación RGB en ocho bits. La distribución de los colores es la siguiente: los tres bits más significativos son de rojo (7:5), los tres siguientes (4:2) son de verde, y los dos menos significativos (1:0) son de azul.

```
entity ROM_RGB_8b_pacman10x10 is
  port (
    clk  : in  std_logic;
    addr : in  std_logic_vector(7-1 downto 0);
    dout : out std_logic_vector(8-1 downto 0)
  );
end ROM_RGB_8b_pacman10x10;

architecture BEHAVIORAL of ROM_RGB_8b_pacman10x10 is
  type memostruct is array (natural range<>) of std_logic_vector(8-1 downto 0);
  constant filaimg : memostruct := (
 --"RRRGGGBB"
   "00000000", "00000000", "01001000", "10110100", "11111100", "11111100", "11011000",
   "10010000", "00100100", "00000000", "00000000", "10010000", "11111101", "11111101",
   "11111100", "11111100", "11111100", "11111101", "11011001", "01001000", "01001000",
   "11111101", "11111101", "11111100", "11111100", "11111100", "11111101", "11111101",
   "11111110", "11011001", "10111000", "11111101", "11111100", "11111100", "11111100",
   "11111100", "11111101", "11011001", "01101000", "00000000", "11111100", "11111100",
   "11111100", "11111100", "11111100", "11011000", "01101100", "00100100", "00100000",
   "00000000", "11111100", "11111100", "11111100", "11111100", "11111100", "11011000",
   "01101100", "00000000", "00000000", "00000000", "10111000", "11111101", "11111100",
   "11111100", "11111100", "11111100", "11111100", "11111101", "10010000", "00000000",
   "01001000", "11111101", "11111101", "11111100", "11111100", "11111100", "11111100",
   "11111101", "11111101", "11011001", "00000000", "01101100", "11111101", "11111101",
   "11111100", "11111100", "11111101", "11111101", "11011001", "01001000", "00000000",
   "00000000", "01001000", "10110100", "11111100", "11111100", "11111100", "10010000",
   "00100100", "00000000");
begin
  P_ROM: process (clk)
  begin
    if clk'event and clk='1' then
      dout <= filaimg(to_integer(unsigned(addr)));
    end if;
  end process;
end BEHAVIORAL;
```

Código 12.8: Memoria ROM de 10x10 y 256 colores[102]

De esta manera tenemos 256 colores y podemos aprovechar mejor la memoria con las capacidades de la *Nexys2*. También para la *XUPV2P* puede ser conveniente porque no siempre es necesario mostrar los más de 16 millones de colores que se guardan con 24 bits. De esta manera se ahorra memoria[103].

[102] El código de esta memoria se puede descargar de la referencia [28web]

[103] Quizá, como la *XUPV2P* puede mostrar más colores, sería más conveniente usar nueve bits, utilizando el bit de paridad de las BRAM como el tercer bit del azul, ya que de cualquier manera se desperdiciaría al emplear la configuración de ocho bits (recuerda la tabla 12.1)

Ahora las asignaciones de los colores serán diferentes. En el código 12.9 se muestran las asignaciones para la *XUPV2P*. Estas asignaciones están dentro de un proceso secuencial y por eso se puede asignar dos veces la misma señal. Como sabemos, cuando se asigna una señal dos veces en un proceso, la señal recibe el último valor asignado. La primera asignación es para asignar a los bits menos significativos el valor del bit más significativo de `datmem` (también se podía haber asignado un cero o un uno). En la segunda asignación se asignan los bits más significativos y se sobreescribe la anterior asignación de estos bits. En el código 12.10 se muestra la asignación para la *Nexys2*.

```
...
 red               <= (others=>datmem(7));
 green             <= (others=>datmem(4));
 blue              <= (others=>datmem(1));
 red  (7 downto 5) <= datmem(7 downto 5);
 green(7 downto 5) <= datmem(4 downto 2);
 blue (7 downto 6) <= datmem(1 downto 0);
...
```

Código 12.9: Asignación del valor del dato de la memoria de 8 bits de color a los colores de la XUP (en proceso secuencial)

```
...
 red   <= datmem(7 downto 5);
 green <= datmem(4 downto 2);
 blue  <= datmem(1 downto 0);
...
```

Código 12.10: Asignación del valor del dato de la memoria de 8 bits de color a los colores de la Nexys2

Si implementas estos circuitos deberá salir el muñeco del videojuego *Pacman* . Saldrá muy pequeño, ya que sólo tiene diez píxeles de ancho y alto. Se ha escogido un tamaño tan pequeño para poder incluir el código VHDL de las ROM en este libro.

12.2.6. Memorias ROM grandes: convertir imágenes

Los ejemplos de imágenes que hemos visto se corresponden con imágenes pequeñas. De hecho, estas imágenes han sido tan pequeñas que el sintetizador no ha utilizado las BRAM sino que las ha guardado en la lógica distribuida de la FPGA. Una imagen pequeña como la del código 12.8 tiene cien elementos pero casi no se ve en la pantalla. Pasar manualmente una imagen mayor a una ROM descrita VHDL es un trabajo tedioso y estéril. En la página web de este libro [28web] hay un programa que convierte una imagen en una ROM descrita en VHDL.

En ese enlace también hay memorias ROM de mayor tamaño descritas en VHDL por si no puedes convertir las imágenes y quieres hacer pruebas. Prueba a dibujar en la VGA una imagen de mayor tamaño (por ejemplo 100x100) guardada en una ROM descrita en VHDL.

12.3. Memorias RAM

Una memoria RAM se puede describir de muchas maneras en VHDL. Para las FPGAs que estamos usando, *Xilinx* propone varios modelos de descripción para usar las BRAM. Las descripciones cambian según el número de puertos, los modos de funcionamiento y si se quieren usar las BRAM o la lógica distribuida. En este libro sólo se describirán dos tipos de descripción de memorias RAM, en la referencia [30xst] se pueden consultar todas las descripciones propuestas.

En el código 12.11 se muestra una memoria de doble puerto. El puerto "A" es de escritura y lectura, para escribir hay que activar el *write enable* (`wea`). El puerto "B" es sólo de lectura. Si quisiésemos una memoria de un único puerto, bastaría con quitar los puertos `addrb` y `doutb` y las sentencias relacionadas con ellos. En este ejemplo se ha hecho una memoria de noventa mil posiciones y ocho bits de ancho de palabra. La dirección de memoria tiene 17 bits, ya que 2^{17} es la primera potencia de dos superior (o igual) a 90000.

```
entity BRAM_DUAL is
  port(
    clk          : in  std_logic;
    wea          : in  std_logic;
    addra        : in  std_logic_vector(17-1 downto 0);
    addrb        : in  std_logic_vector(17-1 downto 0);
    dina         : in  std_logic_vector(8-1 downto 0);
    douta        : out std_logic_vector(8-1 downto 0);
    doutb        : out std_logic_vector(8-1 downto 0)
  );
end BRAM_DUAL;

architecture BEHAVIOURAL of BRAM_DUAL is
  type memostruct is array (natural range<>) of std_logic_vector(8-1 downto 0);
  signal memo          : memostruct(0 to 90000-1);
  signal addra_int,    addrb_int     : natural range 0 to 2**17 -1;
  signal addra_rg_int, addrb_rg_int  : natural range 0 to 2**17 -1;

begin
  addra_int <= TO_INTEGER(unsigned(addra));
  addrb_int <= TO_INTEGER(unsigned(addrb));

  P: process (clk)
  begin
    if clk'event and clk='1' then
      if wea = '1' then   -- si se escribe en a
        memo(addra_int) <= dina;
      end if;
      addra_rg_int <= addra_int;
      addrb_rg_int <= addrb_int;
    end if;
  end process;

  doutb <= memo(addrb_rg_int);
  douta <= memo(addra_rg_int);
end BEHAVIOURAL;
```

Código 12.11: Memoria RAM de doble puerto, uno de escritura y lectura (modo write first) y el otro de sólo lectura

Esta memoria es de tipo *write first* (se escribe primero y luego se lee lo que se ha escrito), que significa que se cuando se escribe un dato (`wea='1'`), en el siguiente ciclo de reloj la memoria devuelve el dato que se está introduciendo por `douta`, es decir no devuelve el dato que había en la posición de memoria antes de la escritura.

El otro caso es que la memoria leyese el dato antes de escribir: *read first* (primero se lee y luego se escribe). El código 12.12 muestra el proceso si queremos que la memoria sea *read first* (primero lectura). Para algunas FPGAs, este tipo de memoria no siempre se sintetizan usando BRAM, por lo que debes de mirar el informe de síntesis. El resto de la arquitectura sería similar al del código 12.11.

```
...
  P: process (clk)
  begin
    if clk'event and clk='1' then
      if wea = '1' then   -- si se escribe en a
        memo(addra_int) <= dina;
      end if;
      douta <= memo(addra_int);
      doutb <= memo(addrb_int);
    end if;
  end process;
...
```

Código 12.12: Proceso de la memoria RAM de doble puerto, uno de escritura y lectura (modo read first) y el otro de sólo lectura

El código 12.11 no está descrito con constantes para poner las dimensiones de la memoria con un ejemplo concreto, pero sería más conveniente haberlo descrito con genéricos o constantes declaradas en los paquetes.

Hay maneras válidas de decribir una memoria en VHDL que el sintetizador de *Xilinx* no reconoce como descripciones que se puedan *mapear* en una BRAM. Si este fuese el caso, el sintetizador podría estar mucho tiempo intentando *mapear* la memoria en la lógica distribuida de la FPGA. Si ves que está sintetizando durante mucho tiempo:

- Revisa la descripción de la memoria
- Mira el informe de síntesis (apartado 8.5.1), comprobando los recursos de la FPGA utilizados, verificando si está usando los BRAM o la lógica distribuida
- Comprueba que el tamaño de la memoria no supera la cantidad de BRAM de tu FPGA (tabla 12.2)
- Prueba a realizar memorias más pequeñas y comprueba los recursos utilizados en ellas

12.4. Uso de memorias usando el Coregen *

Una alternativa a describir la memoria en VHDL es utilizar el *Coregen* de *Xilinx*. El *Coregen* permite crear núcleos (*cores*) de módulos sin tener que describirlos en VHDL. Esto por un lado es una ventaja, aunque por otro, puede hacer más complicada la visibilidad de la simulación. Aún así, lo veremos en este apartado para saber que existe la posibilidad y además así podemos ver las distintas posibilidades de las memorias con los BRAM.

En el proyecto que tengamos del *ISE* abrimos la ventana de creación de una nueva fuente (*Create New Source*), y seleccionamos *IP* (*Coregen & Architecture Wizard*). IP es el acrónimo de *Intellectual Property*, que se refiere a un componente que se puede utilizar bajo ciertas condiciones legales.

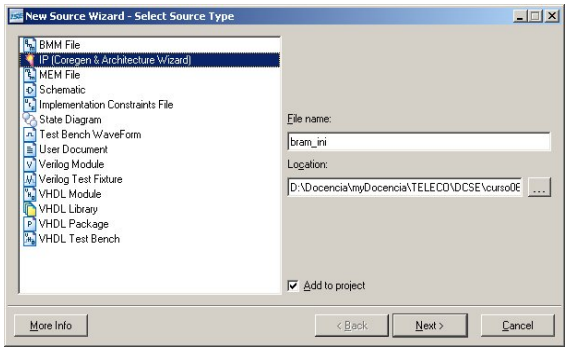


Figura 12.11: Creación de un IP

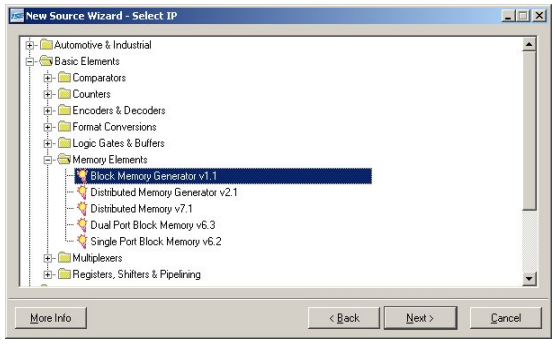


Figura 12.12: Selección de Block Memory Generator

Después de pinchar en *Next* aparecerá un menú de directorio para seleccionar el tipo de IP que queremos implementar. Entramos en *Basic Elements* → *Memory Elements* y seleccionamos *Block Memory Generator* (las opciones de *Dual* y *Single Port Block Memory* no se deben usar a no ser por motivos de compatibilidad con diseños anteriores).

En la siguiente ventana pinchamos en finalizar, y posteriormente nos aparecerá una ventana del generador de memoria, en la cual tendremos que introducir las especificaciones de la nuestra. En la primera ventana pondremos el nombre del componente y seleccionamos el tipo de memoria. La memoria (`bram_ini`) la haremos "*Single Dual Port*". Esta opción permite escribir desde el puerto `A` y leer por el puerto `B`. Otra opción ("*True Dual Port RAM*") permite leer y escribir desde los dos puertos. Elegiremos "*Single Dual Port*" que es como la RAM del código 12.11, que tiene un puerto de escritura y lectura, y el otro puerto de sólo de lectura.

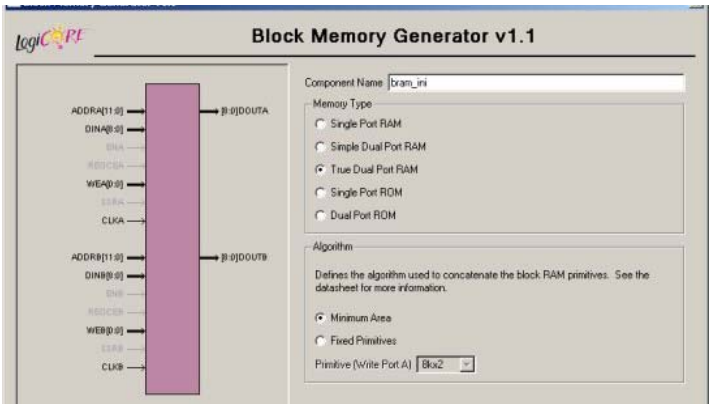


Figura 12.13: Selección de opciones de la memoria (I)

En la siguiente ventana (figura 12.14) seleccionamos el ancho de las palabras (8 bits) y la profundidad de la memoria (90000→300x300). El modo de operación (*Write First*) y el resto de opciones las dejamos igual.

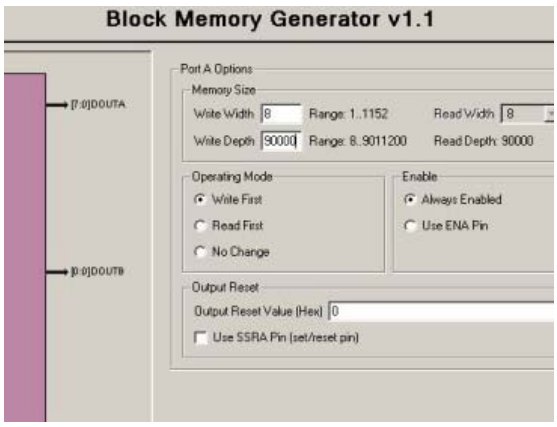


Figura 12.14: Selección de opciones de la memoria (II)

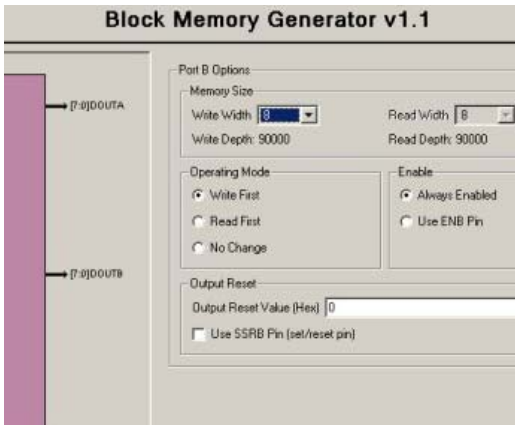


Figura 12.15: Selección de opciones de la memoria (III)

En la siguiente ventana (figura 12.15) aparecerán las opciones para el puerto B, en ella ponemos el *Write Width* a 8 (el mismo que para el A), verificamos que la profundidad salga la misma. En el modo de operación (*Operating Mode*) se indica qué dato se obtiene de lectura cuando se está escribiendo, así que este modo sólo afecta cuando se está escribiendo. Se puede seleccionar que sea:

- De primero escritura (*Write First*): se obtendrá como dato de lectura el mismo que se escribe.
- De primero lectura (*Read First*): se obtendrá como dato de lectura el que había antes de escribir.
- Sin cambio (*No Change*): se obtendrá como dato de lectura el último que se leyó antes de empezar la escritura.

El resto de ventanas que aparezcan se dejarán con los valores por defecto. Para terminar se pincha en *finish* en la última ventana (nº 5), y la herramienta dedicará cierto tiempo a generar el componente.

La memoria generada será similar a la descrita en el código 12.11. El cronograma de lectura y escritura de la RAM se muestra en la figura 12.16 (en nuestro caso hemos creado la memoria sin señal de habilitación `ENA`). El modo de la memoria de este cronograma es de primero escritura, por lo que si se está escribiendo (`WEA='1'`), el dato que se obtiene en `DOUTA` es el mismo que se está escribiendo (`DINA`).

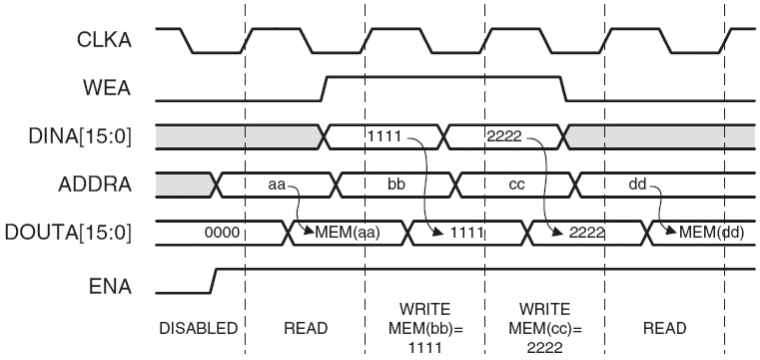


Figura 12.16: Cronograma de con la lectura y escritura de la BRAM (modo write first)

En la figura 12.17 se muestra un cronograma de una memoria con modo de primero lectura.

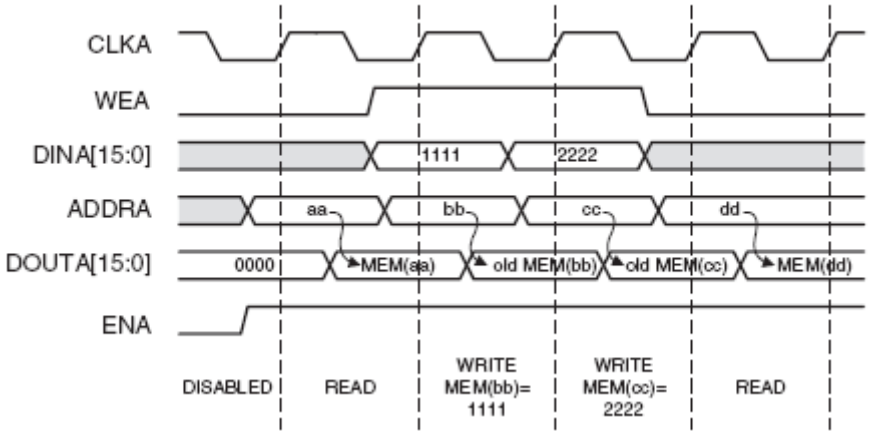


Figura 12.17: Cronograma de con la lectura y escritura de la BRAM (modo read first)

12.5. Guardar y mostrar una imagen que llega por el puerto serie

Para terminar proponemos un diseño en el que enviemos una imagen de un tamaño determinado desde nuestro ordenador por el puerto serie a la FPGA y la mostremos por pantalla. Para simplificar el circuito, previamente hay que determinar el tamaño de la imagen, el número de colores, la profundidad de color y la velocidad de transmisión de la UART. El diagrama de bloques de un diseño que implementa esta funcionalidad se muestra en la figura 12.18.

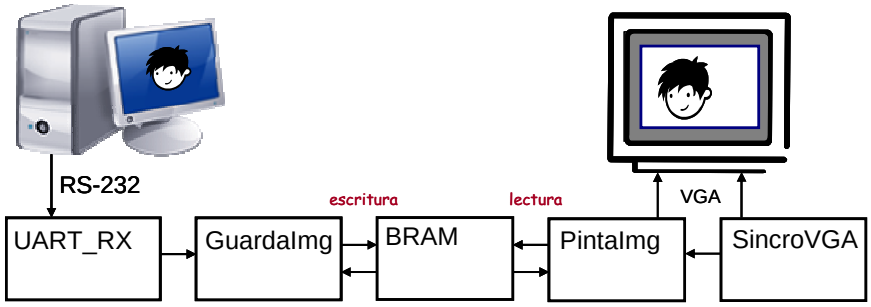


Figura 12.18: Diagrama de bloques de la práctica

Te recomendamos que para empezar realices primero la parte de la derecha del circuito (bloques `BRAM`, `PintaImg` y `SincroVGA`) y compruebes que funcione bien. Esto se haría con una ROM que tendrá el mismo tamaño de la RAM que usaremos para guardar la imagen de la UART. La ROM guardará la imagen que se va a enviar por la UART[104]

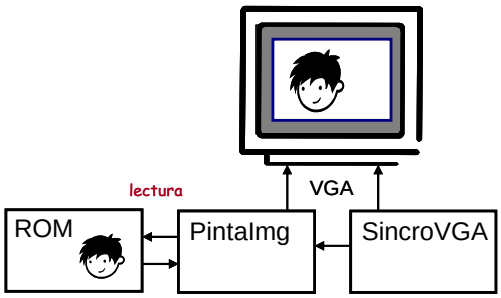


Figura 12.19: Primera versión del circuito

Una vez que la parte derecha funcione bien, cambiaremos la ROM por una RAM de doble puerto. El puerto de escritura "A" estará conectado con el bloque `GuardaImg`, e irá guardando cada píxel que llega desde la UART. Este bloque irá aumentando la dirección de memoria cada vez que llega un nuevo dato de la UART. El puerto de lectura "B" estaría conectado con `PintaImg`, y funcionaría exactamente igual que la primera versión (figura 12.19).

A continuación veremos en qué formato enviaremos la imagen y cómo la enviaremos.

12.5.1. Conversión de la imagen

Hay multitud de formatos para guardar una imagen, muchos de ellos comprimen la imagen para reducir su tamaño. A nosotros nos interesa tener la imagen en formato *raw* (en crudo), esto es un formato binario sin comprimir y sin cabeceras, en la que los píxeles están consecutivos. Si la imagen está en escala de grises, cada byte corresponderá con un píxel y si está en color, cada píxel tiene tres bytes correspondientes con los colores rojo, verde y azul. Este es el formato más sencillo aunque los tamaños de las imágenes son grandes por no usar compresión.

Podríamos realizar un circuito que incluya la conversión de una imagen en un formato habitual (*jpeg, png, gif*) a *raw*, in embargo esto complicaría mucho el circuito. Así que será más sencillo enviar la imagen directamente en formato *raw*.

Para convertir una imagen a formato *raw* podemos usar los programas gratuitos *Irfanview* [15irfan] o *ImageMagik* [14imag]. En la conversión, las imágenes *raw* las guardamos sin cabeceras (hay algunos programas que permiten cabeceras en los ficheros *raw*). Si las imágenes son en color, las guardamos entrelazando los colores RGB, es decir que primero va el píxel rojo, luego el verde y finalmente el azul del primer píxel, posteriormente va el segundo píxel en la misma sucesión de colores y así hasta el final.

Como las imágenes *raw* no tienen cabeceras ni ninguna otra información, al abrirlas con un programa de dibujo tenemos que darle las dimensiones y las características del color (cuantos bytes de color por píxel). Por eso es conveniente que en el nombre de la imagen incluyas sus dimensiones y el número de colores.

[104] Puedes usar el programa de la referencia [28web]

La imagen *raw* se guardará según los colores definidos de la imagen. Si la imagen está en escala de grises se guardará con ocho bits por píxel, y si está en color se guardará con 24 bits por píxel. Sin embargo esto a veces puede llevar a confusión ya que una imagen puede tener sólo colores en escala de grises, pero estar definida con 24 bits de color. Al guardarla en formato *raw* podría suceder que no la estuviésemos guardando con los colores que pensamos.

Por eso es importante verificar el tamaño de la imagen una vez creada. Las propiedades del archivo te indicarán el número de bytes de la imagen. Por ejemplo, si tienes una imagen de 10x10, tendrá 100 píxeles. Si esta imagen tiene un tamaño de 300 bytes, será que tiene 3 bytes por píxel, por lo que está a color (24 bits). Si por el contrario el tamaño de la imagen es de 100 bytes, implica que sólo tiene un byte por píxel, por lo que está en escala de grises.

Si no quieres tener problemas en la conversión a *raw*, te recomendamos que antes de convertir una imagen a *raw* la conviertas primero a:

- PGM: si la quieres en escala de grises (1 byte: 8 bits)
- PPM: si la quieres a color (3 bytes: 24 bits)

Y luego conviertas una de estas dos a *raw*. Esto te da la seguridad de saber siempre cuantos bytes se están usando para cada píxel. Además, a partir del formato PGM y PPM puedes convertir la imagen a una ROM en VHDL usando el programa disponible en [28web]

12.5.2. Envío de la imagen

Si queremos hacer el circuito lo más sencillo posible, será mejor enviar la imagen en escala de grises. Más tarde podremos modificar el diseño, enviando una imagen en color (3 bytes por píxel) y guardando los tres colores en un byte, como vimos al final del apartado 12.2.5. También se podrá guardar la imagen en color en tres memorias de un byte cada una.

Para enviar la imagen desde el ordenador puedes utilizar el programa *RealTerm* [24realt], que en la pestaña *Send* tiene un botón llamado *Send File* para enviar ficheros. Si la velocidad de transmisión es muy baja (9600 baudios) el envío de la imagen completa tardará bastante. Una velocidad de envío de 115200 es razonable. Recuerda que la velocidad de envío del ordenador debe ser la misma que la velocidad del receptor de la UART que has implementado.

Si ves que no se muestra bien la imagen desde la FPGA, prueba a realizar el envío con dos bits de parada (*Port -> Stop Bits*), ya que puede ser que se estén enviando los bytes muy seguidos unos de otros. Si esto fuese así, tendrías que cambiar el estado del bit de fin del receptor para que no dure tanteo.

Si sigue sin funcionar, comprueba que la frecuencia de envío de la UART coincide. Cambia esto en la pestaña *Port -> Baud* y pincha en el botón *Change*, porque si no, no se actualiza. Comprueba en la parte inferior de la ventana que la velocidad es la correcta. Si aún así no sale, y has seguido todos los pasos (haciendo el circuito simplificado con la ROM de la figura 12.19) tendrás que simular.

13. Mostrar caracteres por pantalla

En esta práctica mostraremos por pantalla los caracteres que se escriban desde el teclado. Para ello, necesitaremos tener una memoria ROM en la que tendremos el mapa de bits de los caracteres. En una versión inicial, el circuito simplemente se encargará de mostrar por pantalla los caracteres introducidos. Posteriormente se podrán incluir acciones como el borrado de caracteres y el retorno de carro.

Para facilitar el diseño usaremos caracteres del mismo tamaño, de modo que cada carácter tiene reservado el mismo espacio. Esto nos permitirá dividir el espacio de nuestra pantalla en una especie de tablero de ajedrez. En cada casilla de este tablero podrá ir cualquier caracter. Esto hará reducir la complejidad del circuito. Así, por ejemplo, la letra "A" ocupará lo mismo que la letra "i", y ambas podrán ocupar la misma casilla.

Asimismo, cada casilla del tablero formará una cuadrícula en la que cada celda de esta cuadrícula representa un píxel. Las celdas de esta cuadrícula se pintarán de tal manera que formen el caracter deseado. Todas estas cuadrículas tendrán el mismo tamaño. Para ayudar a comprender esto, la figura 13.1 muestra ejemplos de las cuadrículas (mapas de bits) para las letras "A", "B" e "i". Observa que todos los caracteres están inscritos en una cuadrícula de 8x16 píxeles. Como nuestra pantalla es de 640x480, podremos escribir 80x30 caracteres.

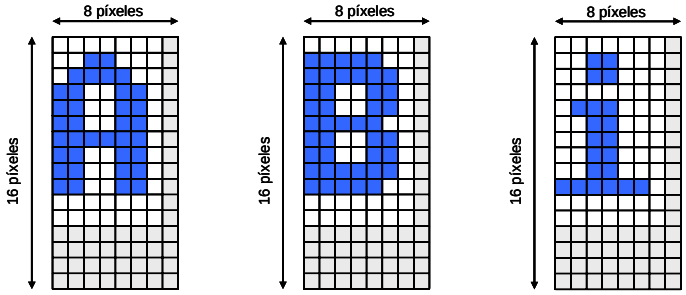


Figura 13.1: Mapa de bits de la "A", la "B" y la "i"

Un mismo carácter puede dibujarse de muy diferentes maneras, tanto más cuanto mayor sea el tamaño de la cuadrícula, y es por eso que hay tantos tipos de fuentes. Además, normalmente un carácter también se puede escribir en negrita, cursiva, subrayado,... Los mapas de bits de los caracteres se pueden guardar en una memoria ROM (así también lo hacían las impresoras matriciales). También se pueden usar memorias RAM para permitir cargar fuentes distintas.

Un circuito que escribe los caracteres en pantalla se podría realizar como muestra la figura 13.2.

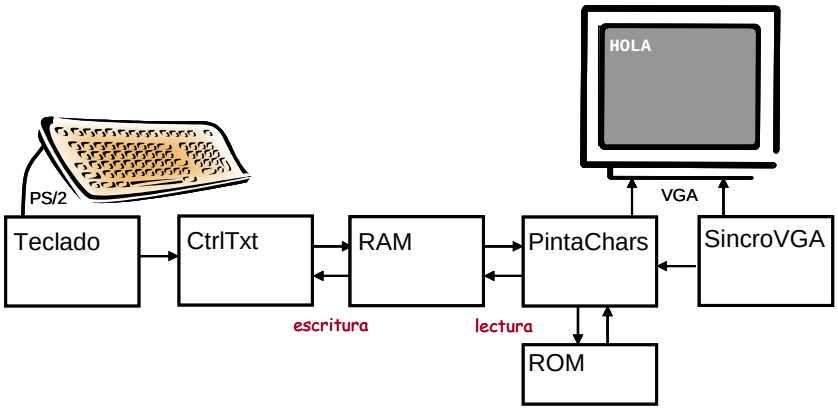


Figura 13.2: Diagrama de bloques del circuito que escribe los caracteres por pantalla

De los bloques de la figura 13.2, el interfaz con el teclado y la sincronización de la VGA ya sabemos cómo se hacen. Como alternativa al teclado se podría emplear el receptor de la UART y escribir desde el *hiperterminal*. A continuación se explicará cómo realizar el resto de bloques.

13.1. Memoria ROM

En la memoria ROM deberían estar los 128 caracteres ASCII (aunque no todos son imprimibles), por tanto, nuestra memoria ROM tendrá 128 cuadrículas similares a las mostradas en la figura 13.1. Si observásemos un fragmento de la memoria ROM tendría el aspecto mostrado en la figura 13.3. El fragmento del ejemplo se corresponde con los caracteres "A" y "B". La parte sombreada se corresponde con un '1' y la sin sombrear un '0'. Este valor nos indicará si tenemos que pintar el píxel de la pantalla en blanco ('1') o en negro ('0'), suponiendo que escribimos los caracteres en blanco sobre fondo negro.

La ROM tiene un ancho de palabra de 8 bits (un byte) para guardar los 8 píxeles de ancho de la cuadrícula, similar a la del apartado 12.2.4 pero con 128 imágenes en vez de con una. La cuadrícula de los caracteres tiene 16 filas de alto, por tanto, cada carácter ocupará 16 posiciones de memoria. Como la ROM tiene que guardar los 128 caracteres ASCII, el tamaño de la memoria será de 128x16=2048 bytes. Para acceder a las 2048 posiciones de memoria necesitamos 11 bits (2^{11} = 2048), por lo tanto, el puerto de la dirección de memoria (`dirmem/addr`) tendrá 11 bits.

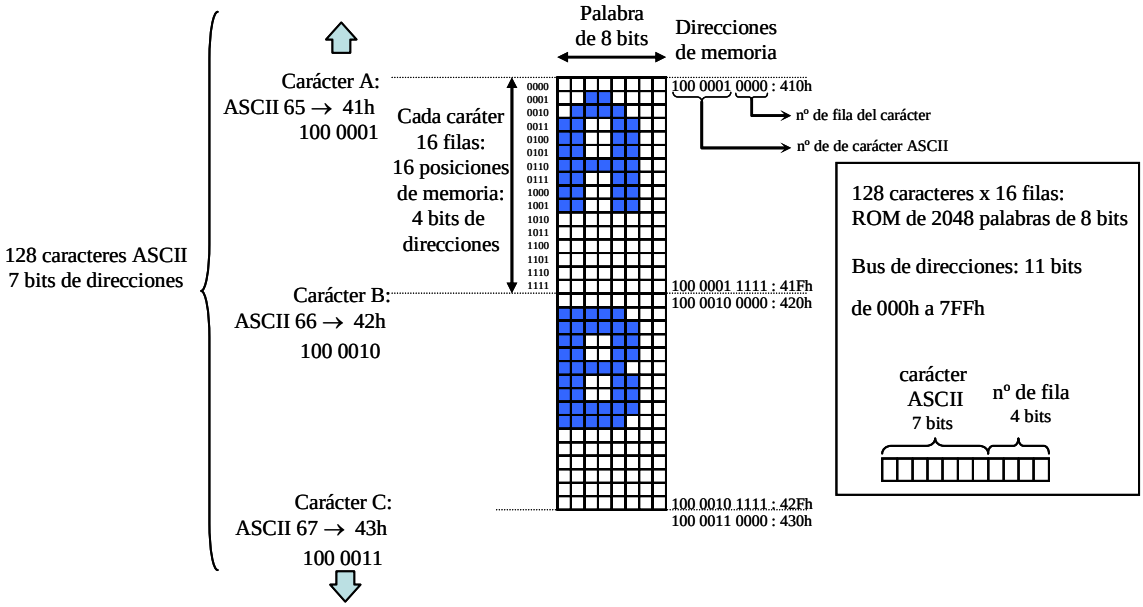


Figura 13.3: Configuración de la ROM para guardar la información de los píxeles de los 128 caracteres ASCII

El hecho de que cada carácter ocupe una cuadrícula de 8x16 es muy conveniente para la implementación del circuito. Estos números son potencias de dos, es decir, números redondos en binario y, por tanto, las divisiones y multiplicaciones se podrán realizar desplazando los bits.

Por esto, de los 11 bits de direcciones que necesitamos para las acceder a las 2048 direcciones de memoria, los siete más significativos indicarán el carácter ASCII al que queremos acceder, mientras que los cuatro menos significativos indicarán la fila de la cuadrícula de dicho carácter. Observa que si el alto de la cuadrícula no fuese una potencia de dos, esto no se podría realizar tan fácilmente (habría que realizar la división).

Finalmente, al acceder a la posición de memoria deseada, la memoria nos devolverá un dato de 8 bits. Dependiendo del bit, escogeremos una columna diferente de la cuadrícula. Esto ya lo hemos visto en el apartado 12.2.4, y cómo evitar que pintemos la imagen simétrica del caracter.

13.2. Memoria RAM

La memoria RAM va a ser una memoria similar a la que usamos en el apartado 12.5. Será una memoria RAM de doble puerto. El puerto "A" conectado al módulo `CtrlTxt` será de escritura y en él se escribirán los caracteres que llegan del teclado. El puerto "B" será de lectura y estará conectado al módulo `PintaChars`, que mostrará por pantalla los caracteres escritos.

La memoria memoria tendrá como mínimo 2400 posiciones (80x30). Cada posición tendrá al menos 7 bits para poder guardar los 128 caracteres ASCII. La memoria se podrá ampliar en caso de que se quiera incluir un *scroll*.

13.3. Control del texto

El módulo de control del texto `CtrlTxt` se encargará de guardar en la memoria RAM los caracteres que llegan del teclado. En una primera versión simplemente realizará esto, en

versiones más avanzadas del diseño se podrá analizar qué tipo de caracteres se están recibiendo y realizar las acciones correspondientes. Por ejemplo, si se pulsa la tecla de borrar, se podría borrar el último carácter escrito, o al pulsar el `Enter` se podría realizar un retorno de carro. En un diseño todavía más avanzado, se podría incluir una línea de comandos en donde se interpreten ciertos comandos preestablecidos.

La pantalla tiene 640 píxeles de ancho y cada carácter tiene un ancho de 8 píxeles, por tanto disponemos de 80 columnas para escribir caracteres. En cuanto a filas disponemos de 30, ya que son 480 píxeles de alto entre los 16 píxeles de alto de cada carácter. Así pues, en total necesitamos 80x30= 2400 caracteres.

Para empezar se puede utilizar una memoria que, cuando esté llena, ya no se puedan añadir más caracteres. Posteriormente se podrá realizar con una FIFO[105], de modo que conforme lleguen nuevos caracteres se vayan incluyendo al final y que si está llena se vayan eliminando los que hayan entrado al principio. Sería conveniente realizarla por filas (80 caracteres), para que eliminase una fila entera. También se podrían hacer de mayor tamaño y permitir la posibilidad de *scroll*.

Así que en su versión más sencilla, este circuito simplemente va a ir escribiendo en la memoria los caracteres ASCII que van entrando desde el teclado. El circuito llevará la cuenta de la posición de la memoria donde debe de entrar el siguiente carácter.

13.4. Bloque que escribe los caracteres en la pantalla

Este bloque recibe del módulo de sincronización la columna y fila que se está mostrando en pantalla (`pxl_num` y `line_num`). A partir de esta columna y fila, el módulo debe realizar los siguientes calculos:

- Calcular la cuadrícula que corresponde con la columna y fila de la VGA
- Hallar el caracter que está escrito en dicha cuadrícula. Esto se hace accediendo a la memoria RAM con el número de cuadrícula
- A partir del caracter que está escrito en dicha cuadrícula y la fila de la VGA, obtener de la memoria ROM el mapa de bits de la fila.
- Seleccionar el píxel de la fila que hay que pintar a partir del número de columna de la VGA

Debido a que hay dos accesos a distintas memorias (la RAM y la ROM), y éstas tienen retardos, se debe prestar especial atención a la sincronización para mostrar el píxel adecuado en cada momento. Como solución, podríamos retrasar dos ciclos de reloj las señales de la VGA[106] como se hizo en el apartado 12.2.3.1. En aquel caso se retrasaron las señales un ciclo de reloj, pero ahora se deben retrasar dos ciclos porque hay dos accesos secuenciales a memorias.

De manera imaginaria, la pantalla VGA estará dividida en cuadrículas de 8x16. Como hemos dicho, resultan 80 columnas x 30 filas, lo que hacen 2400 cuadrículas. Lo primero que tenemos que hacer es averiguar en qué cuadrícula de la VGA estamos según la fila

[105] Del inglés *First In, First Out*, que traducido es *primero en entrar, primero en salir*

[106] Aunque probablemente no sea necesario porque los mapas de bits de los caracteres suelen tener ceros en la primera y última columna, que es donde podría haber problemas. Por tanto, si son todos iguales, no se notaría si estamos pintando el píxel de otro caracter. O si no, saldrían todos los caracteres desplazasos uno o dos píxeles a la derecha. En cualquier caso, lo mejor sería retrasar las señales de la VGA para sincronizarlas

(`line_num`) y la columna (`pxl_num`) de la VGA que estemos. El número de la cuadrícula (`cuad_num`) se corresponde con la numeración de las cuadrículas mostrada en la figura 13.4.

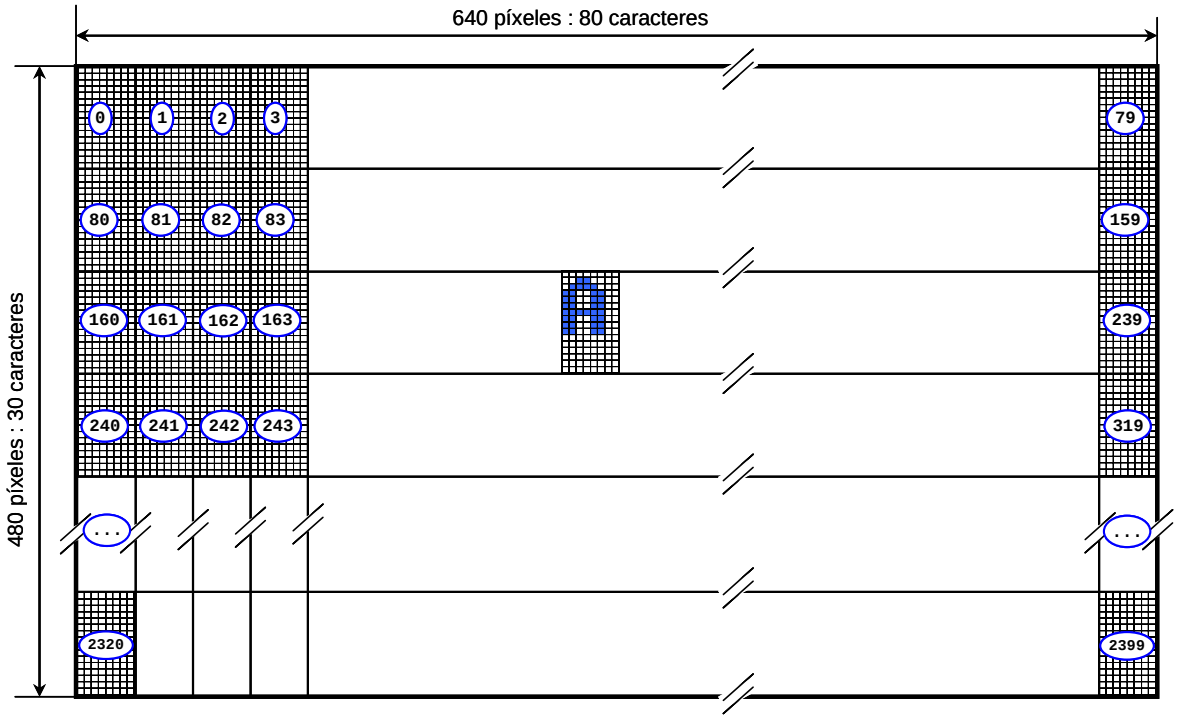


Figura 13.4: División de la pantalla en cuadrículas y numeración de las cuadrículas

Para obtener el número de la cuadrícula tenemos que dividir el número de columna de la VGA (`pxl_num`) entre 8 (ancho de la cuadrícula), con esto tendremos la columna de la cuadrícula (`col_cuad`: entre 0 y 79), y dividir el número de fila de la VGA (`line_num`) entre 16 (alto de la cuadrícula), así tendremos la fila de la cuadrícula (`fila_cuad`: entre 0 y 29). El número de cuadrícula (`cuad_num`) se obtendrá:

$$cuad\_num = (80 \times (line\_num/16)) + pxl\_num/8$$

$$cuad\_num = (80 \times fila\_cuad) + col\_cuad$$

Sin embargo, en el capítulo de operaciones matemáticas (cap. 8) se vio que la multiplicación y sobre todo la división, no son operaciones inmediatas de realizar en hardware. Afortunadamente[107], las divisiones que hay que realizar son entre números redondos en binario, pues 16 y 8 son potencias de 2, y esto lo va a simplificar enormemente.

En la figura 13.5 se muestra cómo se pueden calcular fácilmente la columna y fila de la cuadrícula. Para la columna, como hay que dividir entre 8, basta con no tomar los 3 bits menos significativos. Para la fila habrá que quitar las 4 bits menos significativos. Con esto nos ahorramos la división. Si queremos evitar la multiplicación 80 x `fila_cuad`, podríamos tener un registro que vaya aumentando en 80 cada vez que se pasa a una nueva fila de cuadrícula (similar a como se propuso en el apartado 12.2.3.2). Si no, se puede recurrir a la multiplicación.

[107] No tan afortunadamente, sino que éste ha sido el motivo de escoger esas dimensiones de la cuadrícula

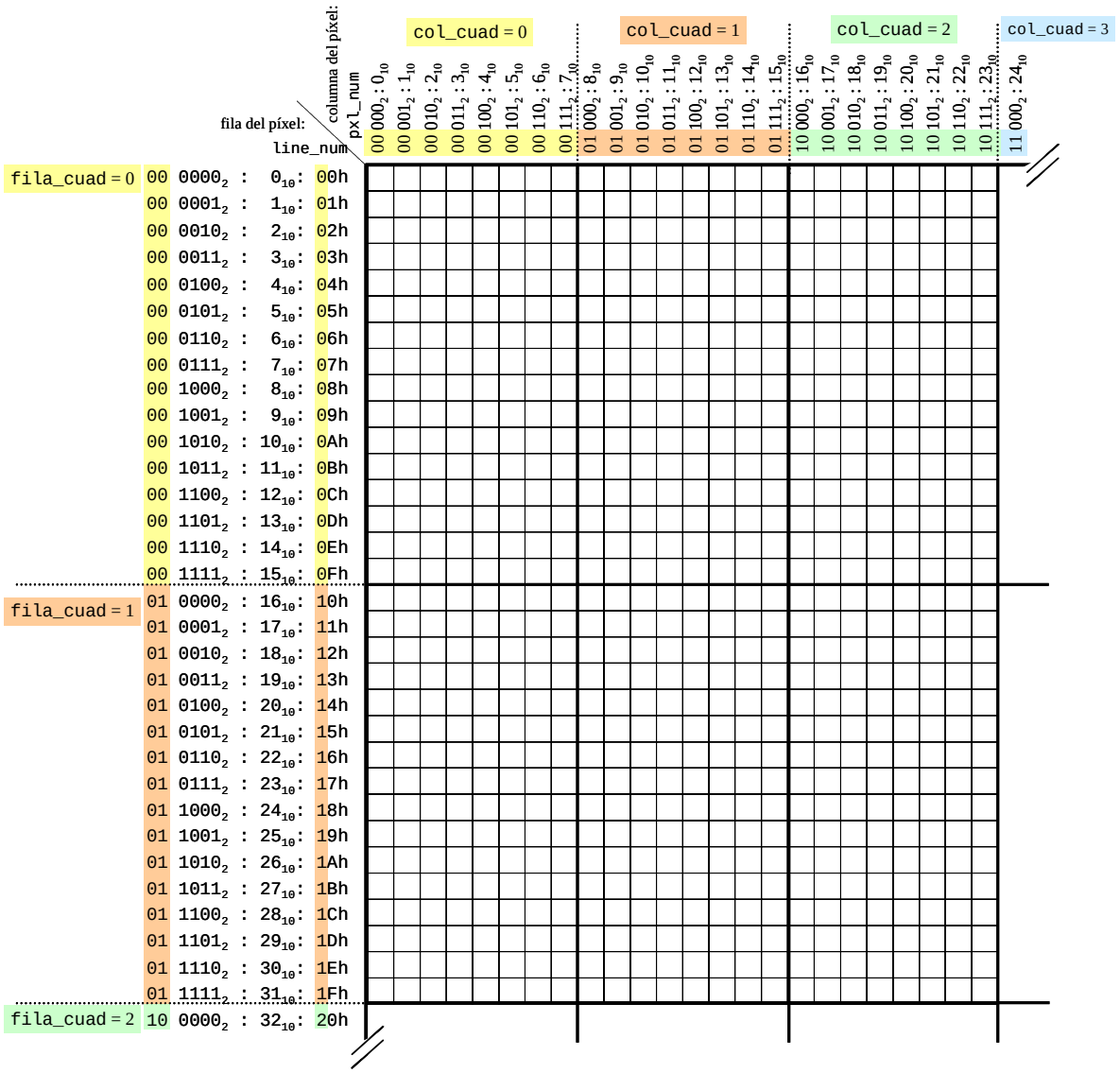


Figura 13.5: Cálculo de la fila y columna de la cuadrícula a partir de la fila y columna de la VGA

Ahora ya tendremos el número de la cuadrícula (`cuad_num`) en la que estamos. Con este índice accedemos a la memoria RAM donde el módulo de control del texto ha guardado los caracteres escritos en el teclado. El acceso a la memoria con el número de cuadrícula nos devolverá el código ASCII del carácter de esa cuadrícula.

Así que ya tenemos los 7 bits de la señal del código ASCII, estos 7 bits son los 7 bits más significativos de la dirección de la memoria ROM donde está el mapa de bits de los caracteres. Recuerda la figura 13.3, los 7 bits más significativos seleccionan el código ASCII.

Los 4 bits menos significativos del bus de direcciones de la ROM indican en qué fila de las 16 filas de la cuadrícula se estamos. Estos 4 bits los tomamos de los 4 bits menos significativos la fila del píxel en que estamos (`line_num`), que son los que no se usaron para calcular la columna de la cuadrícula (figura 13.5).

Ahora ya tenemos los 11 bits de direcciones con los que obtendremos los 8 píxeles de la fila que queremos del carácter. De estos 8 bits seleccionaremos el bit de la columna en que estemos. Esta columna la obtendremos con los 3 bits menos significativos del píxel de la columna (`pxl_num`), estos son los que no habíamos usado para calcular la cuadrícula

(figura 13.5). Recuerda que hay que obtener el complementario del índice para que no salga la imagen simétrica.

Según sea el bit obtenido '0' ó '1' tendremos pintar el píxel negro o blanco en la pantalla. La figura 13.6 muestra el esquema de los cálculos de las direcciones de las memorias para obtener el color del píxel. En estos esquemas y explicaciones hay que tener en cuenta además los rangos resultantes de las multiplicaciones. Por ejemplo, para la multiplicación de `line_num(8 downto 4) x 80` se debe aumentar el tamaño de `line_num(8 downto 4)`, tal como se ha explicado en el apartado 12.2.2 y el código 12.2.

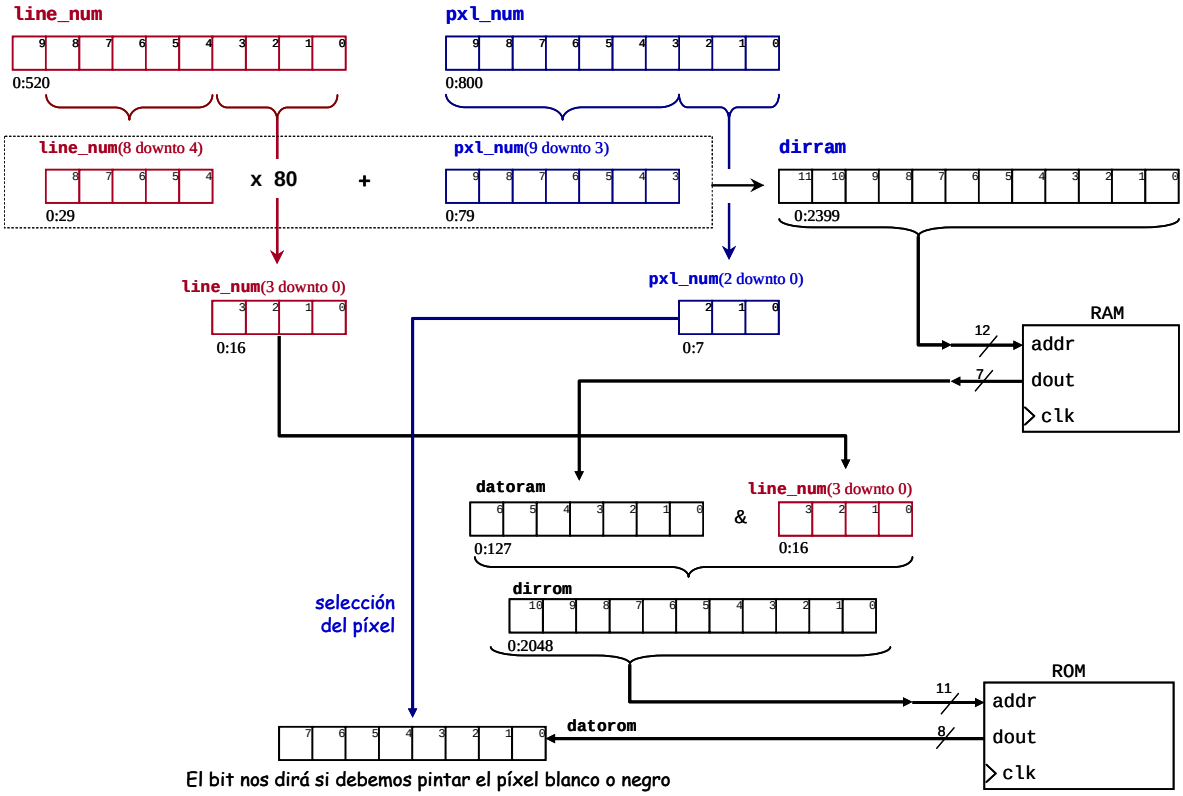


Figura 13.6: Esquema de los cálculos de las direcciones de memoria para la obtención del color del píxel

Con estas explicaciones pensamos que tendrás información suficiente para diseñar el circuito. Si te sale bien, puedes probar a hacer alguna variante de las que se han propuesto. Este tipo de diseños en los que utilizamos las cuadrículas, los emplearemos para el diseño de videojuegos (capítulo 15).

Si tienes problemas en la implementación y no sabes qué hacer para solucionarlos, puedes probar a implementar el circuito con una ROM en vez de la RAM. En la ROM habrá caracteres fijados por tí que se tendrán que mostrar por pantalla. Para este ejemplo, no haría falta que hicieses la ROM con 2400 posiciones de memoria, bastaría con dos o tres filas (unas 200 posiciones sería más que suficiente).

14. Procesamiento digital de imágenes

En esta práctica realizaremos un procesamiento sencillo de una imagen. Primero realizaremos un circuito más sencillo en el que la imagen a procesar estará guardada en una memoria ROM, posteriormente ampliaremos el circuito para que reciba la imagen desde el puerto serie del ordenador. En ambos casos se mostrará en pantalla la imagen original y la procesada.

Antes de analizar el circuito, veremos en qué consiste el procesamiento que queremos realizar.

14.1. Procesamiento de imágenes

A una imagen se le pueden aplicar muchos tipos de procesamientos diferentes. Un procesamiento habitual es aplicar una convolución de ventana 3x3 sobre la imagen original. Estas ventanas tendrán distintos valores según el procesamiento. En la figura 14.1 se muestran los operadores de convolución de tres filtros. En la figura vemos que los filtros de *Prewitt* y *Sobel* tienen dos operadores: el horizontal y el vertical.

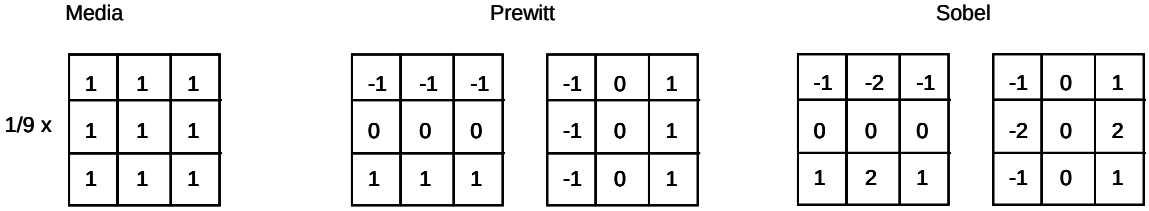


Figura 14.1: Distintos operadores de convolución

Los operadores se van aplicando píxel a píxel y el resultado se asigna al píxel de la nueva imagen. En la imagen 14.2 se muestra cómo se aplicaría el operador horizontal de *prewitt* al píxel (1,1) de la imagen original. La figura incluye la fórmula resultante de aplicar el operador, cuyo resultado será el valor del píxel (1,1) de la imagen procesada.

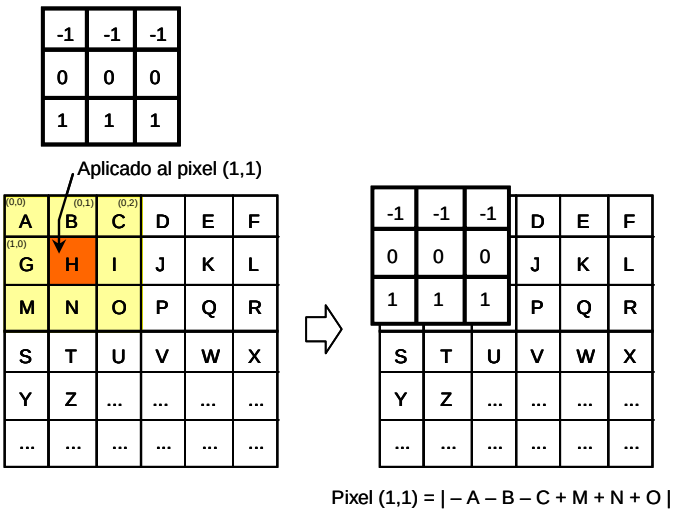


Figura 14.2: Aplicación del operador horizontal de Prewitt al píxel (1,1)

Con el fin de aclarar cómo se realiza el procesamiento, se van a mostrar las ecuaciones que definen los tres operadores de la figura 14.1.

En la figura 14.3 se muestra la numeración de los píxeles para identificarlos en las ecuaciones.

p1	p2	p3
p4	p5	p6
p7	p8	p9

Figura 14.3: Numeración de los píxeles de la ventana

Las ecuaciones del píxel central *p5* son:

Media: $p5_{media} = (1/9) \cdot (p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8 + p9)$

Prewitt: $p5_{prewitt} = |(p7 + p8 + p9) - (p1 + p2 + p3)| + |(p3 + p6 + p9) - (p1 + p4 + p7)|$

Sobel: $p5_{sobel} = |(p7 + 2 \cdot p8 + p9) - (p1 + 2 \cdot p2 + p3)| + |(p3 + 2 \cdot p6 + p9) - (p1 + 2 \cdot p4 + p7)|$

Estas operaciones se deben realizar para todos los píxeles de la imagen excepto para los bits del borde. Los píxeles del borde de la imagen resultante se pueden poner en negro (a cero) para que las imágenes original y procesada tengan el mismo tamaño. Por tanto la imagen resultante tendrá un recuadro negro de un píxel de ancho.

14.2. Circuito que procesa una imagen guardada en una ROM

Queremos implementar un circuito que procese una imagen guardada en una memoria ROM y guarde la imagen procesada en una memoria RAM. El circuito mostrará ambas imágenes por pantalla. Las imágenes estarán en escala de grises con 256 niveles de gris, por lo tanto necesitaremos dos memorias de ocho bits de ancho de palabra. En la tabla 12.3 se anotaron los tamaños máximos de las imágenes de 8 bits/píxel que se podían guardar según el número de imágenes que se tenían. En nuestro caso, como queremos guardar dos imágenes, para la *XUPV2P* podríamos usar imágenes de 350x350 y para la *Nexys2* las imágenes podrían ser de 120x120[108].

El diagrama de bloques de un circuito que haría la funcionalidad deseada se muestra en la figura 14.4.

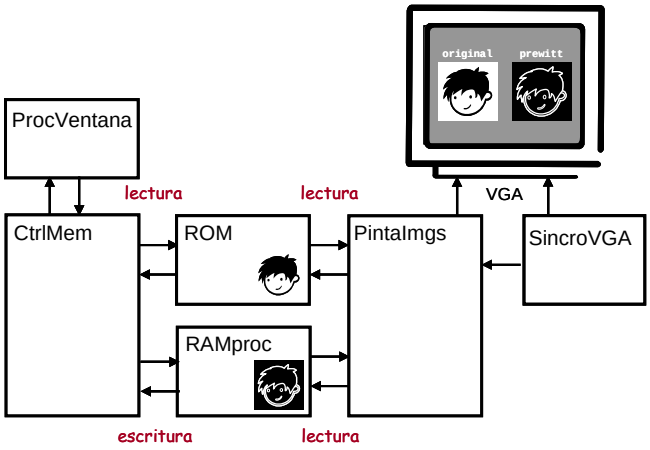


Figura 14.4: Diagrama de bloques del circuito que procesa y muestra una imagen guardada en una ROM

La mayoría de los bloques los sabemos implementar, a continuación se resume su funcionalidad:

- **`SincroVGA`**: el generador de sincronismos de la VGA (apartado 9.3)

[108] Son valores orientativos, no pueden superar los valores de la tabla 12.3 para dos imágenes

- **`ROM`**: memoria ROM con la imagen que queremos procesar. Crearemos la ROM a partir de una imagen en escala de grises (apartado 12.2.6). Para este circuito queremos una memoria ROM de doble puerto, por lo que tendremos que modificarla para añadir el puerto "B" (`addrb` y `doutb`). Los puertos "A" y "B" tendrán el mismo reloj. Es más conveniente que sea de doble puerto para que los accesos a memoria para el procesamiento sean independientes de los accesos a memoria para mostrar la imagen por pantalla.
- **`RAMproc`**: memoria RAM de doble puerto (apartado 12.3). Igual que la memoria ROM, esta memoria también será de doble puerto para independizar la parte que muestra la imagen procesada en la VGA de la parte que guarda la imagen procesada.
- **`PintaImgs`**: es un proceso parecido al que vimos en el capítulo de dibujar imágenes por pantalla (cap. 12), pero con la diferencia de que ahora tenemos que dibujar dos imágenes. Este módulo calculará de manera independiente las direcciones de memoria de ambas imágenes. Por lo tanto, en cada momento se obtendrán dos píxeles: uno de cada memoria. Según la fila y columna de la VGA, el proceso que pinta los píxeles seleccionará qué píxel va a mostrar, que podrá ser el píxel de la imagen original (`ROM`), el píxel de la imagen procesada (`RAMproc`) o el píxel del color del fondo. Las imágenes se mostrarán en distintas zonas de la pantalla, de manera similar a como se hizo en el apartado 12.2.3.3.

  Más adelante se podrá incluir una memoria ROM de caracteres para incluir un texto indicativo de las imágenes, por ejemplo: *Imagen original* e *Imagen procesada Prewitt*.
- **`CtrlMem`**: este módulo se encarga de obtener los píxeles necesarios de la imagen original para poder aplicar las convoluciones de la ventana 3x3 explicadas en el apartado 14.1. Esto se hará para cada uno de los píxeles de la imagen y el resultado se guardará en la memoria RAM (`RAMproc`). Este módulo deberá determinar qué píxeles corresponden con los bordes de la imagen para no aplicar la convolución en ellos y ponerlos directamente a negro en la imagen procesada.
- `ProcVentana`: este módulo recibe de `CtrlMem` los nueve píxeles de una ventana para ser procesados (figura 14.3 del apartado 14.1) y devuelve el valor del píxel resultante después de aplicarle las ecuaciones del procesamiento. Como hay que realizar muchas operaciones, es posible que no dé tiempo realizarlas todas en el mismo ciclo de reloj. Por tanto se recomienda al menos registrar las entradas y las salidas, y analizar si los retardos son lo bastante grandes como para realizar segmentación insertando algún registro más (recuerda el apartado 8.9.1.2).

  Debido a que se han introducido registros, convendría incluir un control para avisar a `CtrlMem` de cuándo se ha terminado la operación (`finproc_vent`) y por tanto, cuándo está disponible el resultado para ser guardado en la memoria RAM. Asimismo, también habría que incluir una señal (`inicproc_vent`) que le indicase a este módulo cuándo están los píxeles preparados para iniciar el procesamiento. Estos controles serán más necesarios en el procesamiento de la media, ya que hay que realizar una división y ésta puede durar varios ciclos de reloj. En la figura 14.5 se muestran las entradas y salidas de este módulo para el caso del procesamiento de *Prewitt* o *Sobel*¸ ya que éstos no necesitan el píxel número cinco.

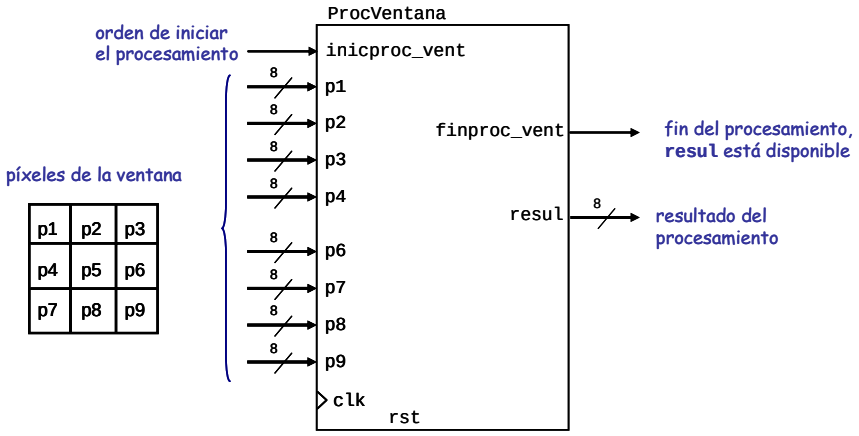


Figura 14.5: Entradas y salidas del módulo `ProcVentana` que realiza la convolución de ventana 3x3

Por último, debes recordar lo explicado en el capítulo 8 de circuitos aritméticos para implementar las operaciones de la convolución (apartado 14.1). Debes tener en cuenta que, al realizar las sumas, el resultado aumentará de rango, y por lo tanto, el resultado total tendrá más de ocho bits. Cuando haya desbordamiento (cuando el resultado sea mayor que 255), el circuito deberá poner el resultado a blanco (255). Si no tienes esto en cuenta, los resultados con valor más alto (más blancos) quedarán con valores muy bajos (cerca del negro) por el truncamiento de los bits más significativos.

En la figura 14.6 se muestra el esquema de las operaciones que hay que realizar para calcular el resultado de la convolución de ventana 3x3 del operador de *Prewitt*. Este esquema opera con números enteros positivos y antes de restar se compara cuál es el mayor, otra alternativa sería realizar las operaciones en complemento a dos. Al final, el resultado es un número de 10 u 11 bits que habría que convertir a ocho bits. Al describir el circuito en VHDL debes de analizar los rangos. También hay que considerar los tiempos y ver si es necesario introducir biestables para segmentar (apartado 8.9.1.2).

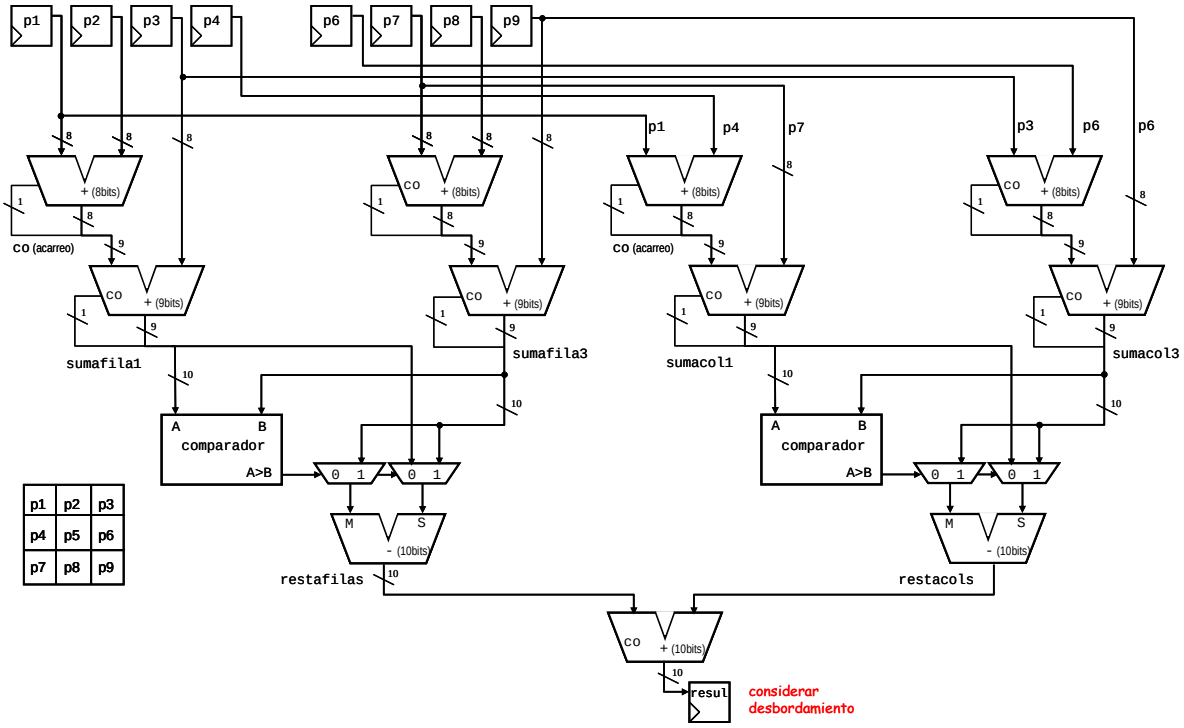


Figura 14.6: Esquema de las operaciones de la convolución 3x3 de Prewitt

14.3. Circuito que procesa una imagen que llega por la UART

Queremos hacer un circuito similar al anterior con la diferencia de que ahora la imagen original la vamos a recibir por la UART en vez de tenerla guardada en la ROM. La figura 14.7 muestra el diagrama de bloques propuesto para el circuito.

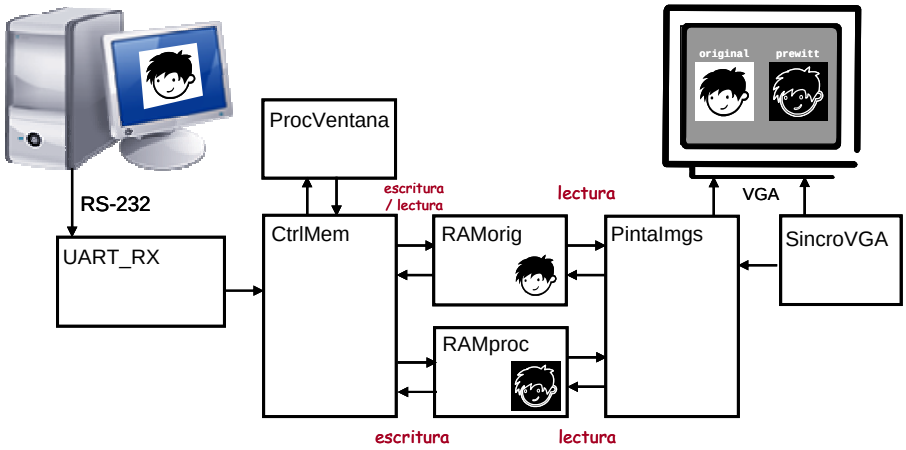


Figura 14.7: Diagrama de bloques del circuito que procesa y muestra una imagen recibida por la UART

Aparte del receptor de la UART, que ya sabemos cómo se implementa, las únicas diferencias con el circuito anterior son:

- **`RAMorig`**: ahora es una memoria RAM en vez de una ROM, tendrá el mismo tamaño, pero se podrá escribir en ella. El bloque `CtrlMem` guardará los píxeles que lleguen de la UART.
- **`CtrlMem`**: este módulo es más complejo que el del circuito anterior. Pues antes de empezar a procesar la imagen debe de haber recibido la imagen desde la UART. Así que deberá de tener unos estados que podrían ser los siguientes:
  - `Espera`: el módulo está a la espera de empezar a recibir los datos de la UART
  - `Guarda`: el módulo guarda los píxeles que llegan de la UART en la memoria `RAMorig`. En este estado actúa de manera similar al módulo `GuardaImg` del apartado 12.5 (figura 12.18)
  - `Procesa`: una vez guardados todos los píxeles de la imagen, comienza a realizar el procesamiento de manera similar al apartado anterior.

Como mejora se podría añadir que el circuito devolviese por la UART la imagen procesada. En este caso habría que incluir un programa en la computadora para guardar la imagen recibida y comprobar que se ha recibido bien.

14.4. Acelerar el procesamiento

Si has implementado y probado el circuito del apartado anterior habrás podido notar que en lo que más se tarda es en la transmisión por la UART y que el procesamiento es mucho rápido. Así que lo que convendría mejorar sería la transmisión, por ejemplo, transmitiendo por un puerto USB o por Ethernet. Sin embargo, estos diseños son más complejos y están fuera de los objetivos de este libro. Por tanto, a pesar de que el cuello de botella no está en el procesamiento, en este apartado veremos cómo acelerarlo.

Propondremos dos mejoras, una consiste en disminuir los accesos a memoria y la otra realiza el procesamiento durante la recepción ("al vuelo").

14.4.1. Disminuir los accesos a memoria

Es probable que ya hayas tenido en cuenta esta mejora al realizar los diseños previos. Cada vez que se realiza el procesamiento de una convolución de ventana 3x3 hay que tener los nueve[109] píxeles de la ventana. Por tanto, habría que realizar nueve accesos a memoria. En nuestro caso no es mucho problema porque las memorias BRAM son muy rápidas, pero no siempre es así.

Sin embargo, como el procesamiento se va realizando con píxeles consecutivos, cuando nos mantenemos en la misma fila, sólo hace falta pedir tres nuevos píxeles, ya que los seis restantes los teníamos de la ventana anterior. La figura 14.8 muestra un ejemplo señalando los píxeles repetidos. Sólo cuando hay un cambio de fila se necesita pedir todos los píxeles.

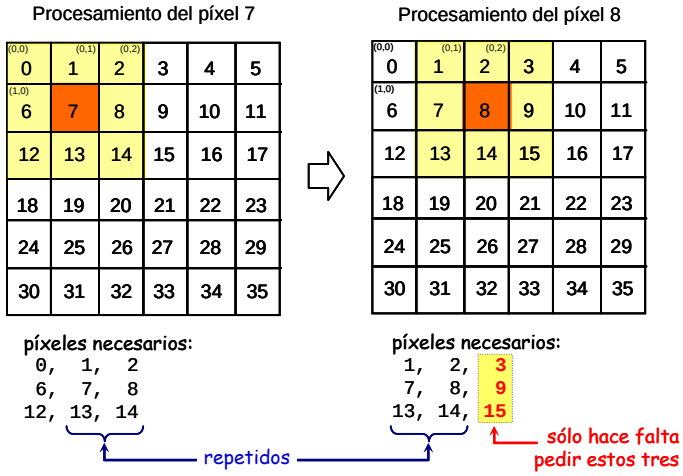


Figura 14.8: Píxeles repetidos en el cálculo de la convolución 3x3 para el procesamiento de dos píxeles consecutivos

Por tanto, aprovechando los píxeles que ya tenemos del procesamiento anterior, conseguimos reducir los accesos a memoria de nueve a tres.

14.4.2. Procesamiento "al vuelo"

El circuito que recibe la imagen de la UART realiza el procesamiento una vez que se ha recibido y guardado la imagen. Una alternativa es realizar el procesamiento al mismo tiempo que se está recibiendo la imagen. Esta opción todavía más interesante en el caso de que no sea necesario guardar la imagen original ya que esta alternativa no sólo realiza el procesamiento más rápidamente sino que además prescinde de la memoria `RAMorig`.

El método consiste en guardar los píxeles que van llegando de la UART en registros de desplazamiento de modo que en cada momento se tengan tres filas de la imagen[110]. La figura 14.9 muestra el esquema de estos registros. Para simplificar se ha escogido una imagen de sólo seis píxeles de ancho. En la figura se puede ver que en la fila de arriba sólo hacen falta tres registros, mientras que las otras dos filas tienen tantos registros como columnas tiene la imagen (en este ejemplo seis).

[109] Nueve píxeles en el procesamiento de la media y ocho píxeles en *Prewitt* y *Sobel*

[110] En caso de que la ventana de la convolución sea de 3x3

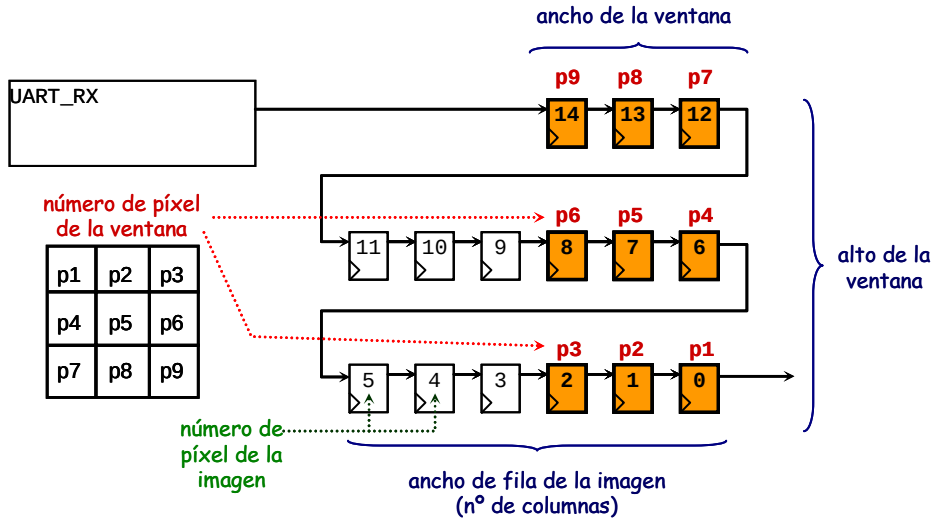


Figura 14.9: Registros que guardan tres filas de la imagen

Podemos ver que los tres registros de la derecha de las tres filas forman los nueve registros que se necesitan para el procesamiento de la ventana.

Para poder empezar a procesar los registros, éstos deben haberse llenado, es decir, el registro `p1` (abajo a la derecha) debe de haber recibido el píxel cero de la imagen. En la figura 14.9 se ha mostrado este caso, en el que ya se han recibido los primeros quince píxeles (de 0 a 14)[111] y ya se puede iniciar el procesamiento. Una vez que se han llenado los registros, los píxeles se seguirán desplazando a la derecha.

En la figura 14.10 se muestra qué sucede cuando llega el píxel número quince. Con la llegada del píxel número quince, el píxel número cero que estaba en el registro de abajo a la derecha se pierde y se reemplaza por el píxel número uno, y así con todos: se desplazan a la derecha con la llegada de cada nuevo píxel desde la UART.

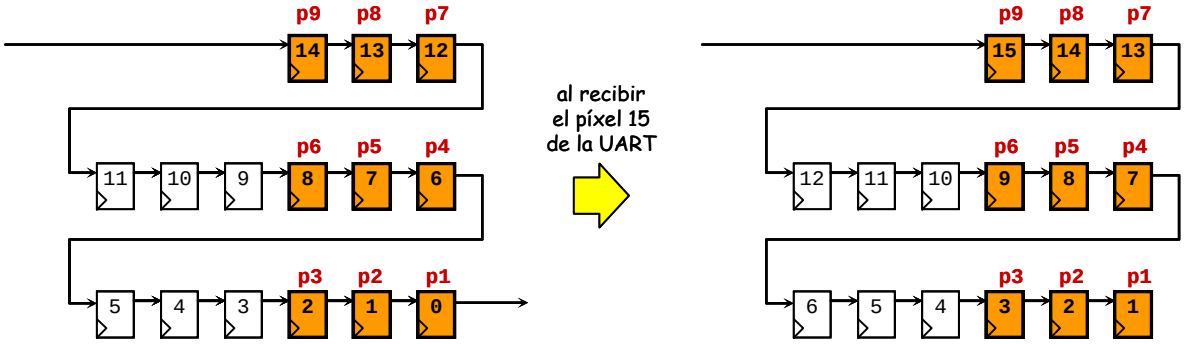


Figura 14.10: Desplazamiento de los registros al recibir un nuevo píxel de la UART

Es interesante observar qué ocurre en los píxeles de los bordes de la imagen. En la figura 14.11 se muestra qué ocurre con los registros cuándo se llega a un píxel del borde. Como se puede observar, la ventana está en columnas no consecutivas y por tanto en estos casos se pondrá como resultado un píxel negro.

[111] Antes de que lleguen los quince píxeles es como si se estuviesen procesando los píxeles del borde, que irán a cero (negro) en la imagen procesada

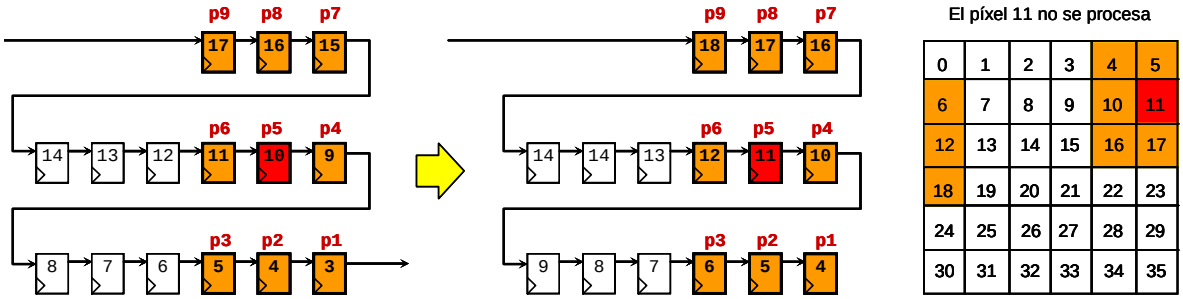


Figura 14.11: Desplazamiento de los registros: llegada de un píxel del borde

Algo similar ocurre cuando se está en un píxel de la primera columna, como se muestra en la figura 14.12. En esta imagen se pasa de estar procesando el píxel número doce, que no se puede procesar por estar en un borde, a procesar el píxel trece, que sí se puede procesar. El píxel que se está procesando coincide con el píxel que está en el registro `p5`, señalado en rojo en las imágenes.

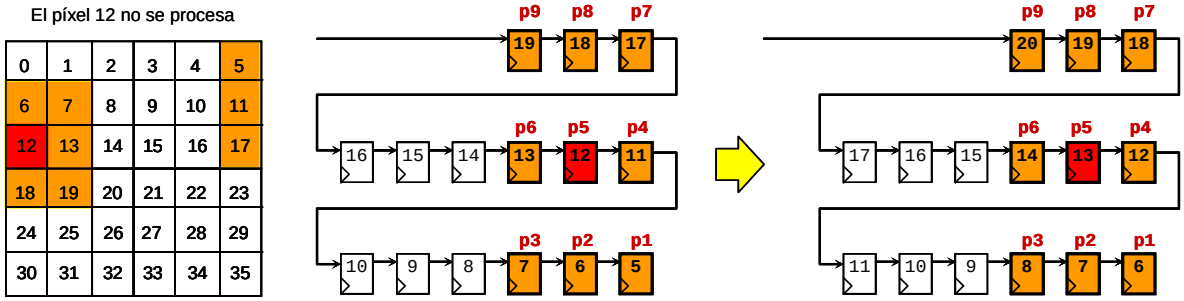


Figura 14.12: Desplazamiento de los registros: salida de un píxel del borde

Cuando se han enviado todos los píxeles (en este caso los 36), se estará procesando el píxel 28, que es el último píxel procesable. A partir de este píxel, el resto de píxeles corresponden con bordes y estarán en negro en la imagen procesada. De todos modos, si se quieran vaciar los registros, habrá que continuar desplazando los registros hasta que se vacíen.

Así que en el circuito hay que incluir un control para indicar si los píxeles corresponden con píxeles del borde, para dar la orden de realizar la convolución o no. En caso negativo, deberá de poner el píxel resultante en negro. Este control deberá contar el número de píxel que se está procesando y determinará también cuándo se termina el procesamiento de la imagen. El control indicará cuándo se deben desplazar los registros, que en líneas generales coincide con la señal de aviso de la UART (`aviso_rx`).

Hay varias formas de implementar este control y el circuito de los registros de desplazamiento. En la figura 14.13 se muestra un esquema simplificado de los bloques de este circuito y su relación con otros módulos. Sin embargo, este esquema es simplificado y puede haber señales que debas incluir, dependiendo de ciertas decisiones de implementación que tendrás que tomar. Por ejemplo, no se ha incluido la señal de habilitación del desplazamiento de los registros.

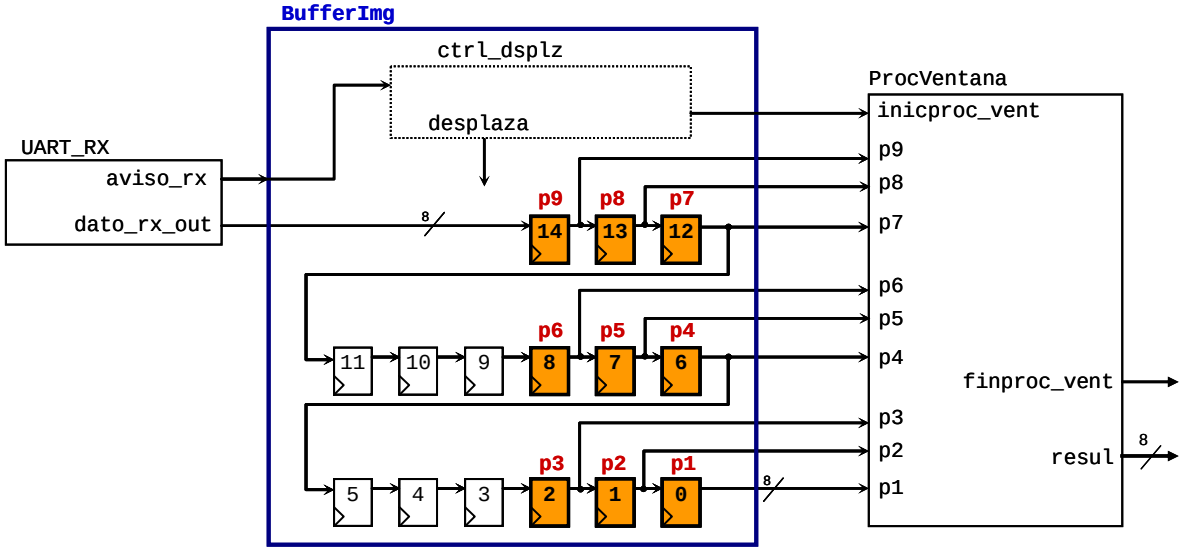


Figura 14.13: Diagrama de bloques simplificado del módulo de los registros de desplazamiento

14.4.3. Uso de un búfer circular

En vez de poner registros de desplazamiento en el circuito anterior, se puede implementar usando búferes[112] circulares. Si la imagen tiene un gran número de columnas, puede ser mejor describir el diseño con búferes circulares en vez de con registros de desplazamiento, ya que el búfer circular se puede implementar con una memoria y podríamos utilizar los bloques BRAM.

Conceptualmente, un búfer circular es una memoria sin comienzo ni fin, es decir, el último elemento es consecutivo con el primero.

No se entrará en detalle con el funcionamiento del búfer circular, pero se darán algunos conceptos de su funcionamiento básico. En la figura 14.14 se han capturado tres momentos en donde se han añadido tres elementos en el búfer. Para facilitar el entendimiento, estos elementos se han llamado consecutivamente con el orden de entrada: 0, 1 y 2.

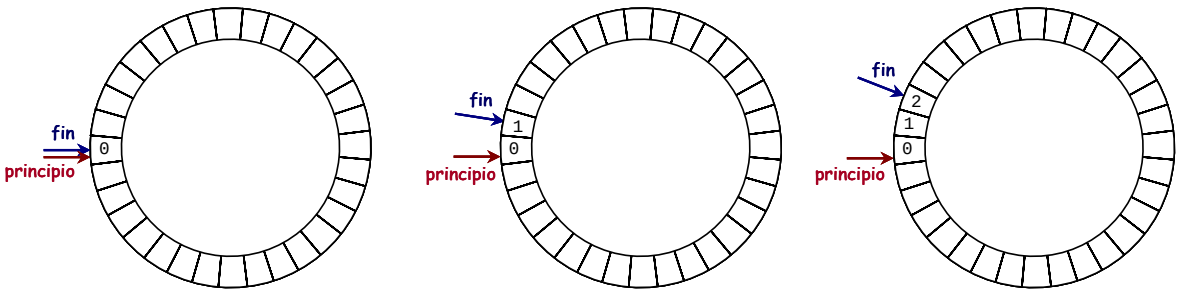


Figura 14.14: Esquema de la escritura de tres elementos en el búfer circular

Normalmente el búfer circular tiene dos punteros que indican el primer elemento del búfer (principio) y el último (fin)[113]. Cada vez que se introduce un nuevo elemento en el búfer se incrementa el puntero de fin, ya que este puntero indica el último elemento introducido.

[112] Adaptación al español de la palabra inglesa *buffer*

[113] Alternativamente, en vez del puntero de fin se puede poner la dirección siguiente, que es el puntero a la dirección donde entrará el nuevo elemento que se introduzca.

Los búferes circulares funcionan como una FIFO. El nombre *FIFO* viene de las iniciales en inglés: *first in - first out*, y son un tipo de pilas en las que el primer elemento que ha entrado es el primero en salir. Si se saca un elemento del búfer saldrá el elemento apuntado por el puntero de principio (que apunta al primer elemento) y el puntero de principio se incrementará.

Si el búfer está lleno y se introduce un nuevo dato se sobreescribe el primer elemento introducido (señalado por el puntero de principio), y los punteros de principio y fin se incrementan una posición (ver figura 14.15).

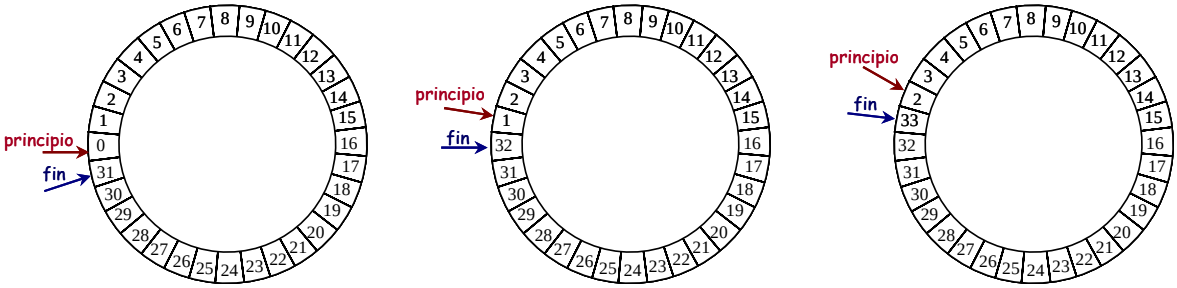


Figura 14.15: Llenado del búfer circular y escritura con el búfer lleno

El búfer circular se puede simplificar en nuestro circuito, ya que se puede trabajar con un único puntero. En nuestro caso no vamos a introducir y sacar elementos en el búfer de forma independiente. Siempre que se introduce un dato nuevo se pondrá en la salida el valor anterior que había en la dirección de memoria donde se escribe.

En la figura 14.16 muestra el movimiento del puntero cuando el búfer no está lleno.

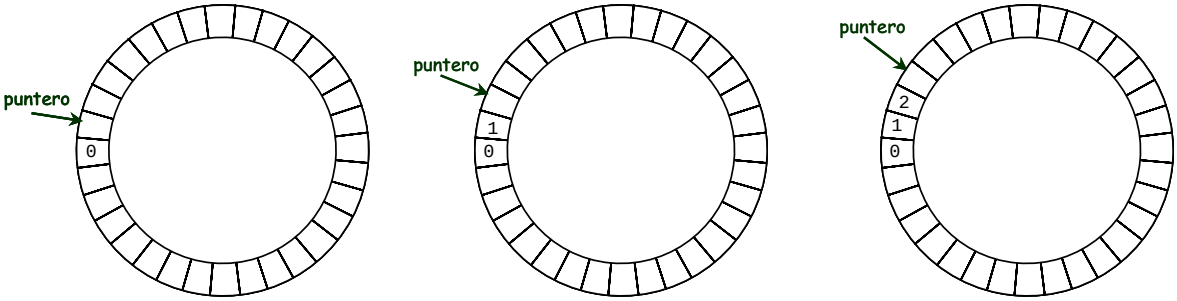


Figura 14.16: Búfer circular con un único puntero

Cuando el búfer está lleno, al introducir un nuevo elemento, deberá sacar el elemento que había. Por ejemplo, en la figura 14.17, cuando se introduce el 32, el búfer saca el 0. Y así sucesivamente, cuando se introduzca un nuevo elemento, el búfer sacará el uno. En realidad, esto es lo que ocurre cuando el búfer no está lleno, pero los valores previos que tenga la memoria son indefinidos (o cero si la memoria se ha inicializado).

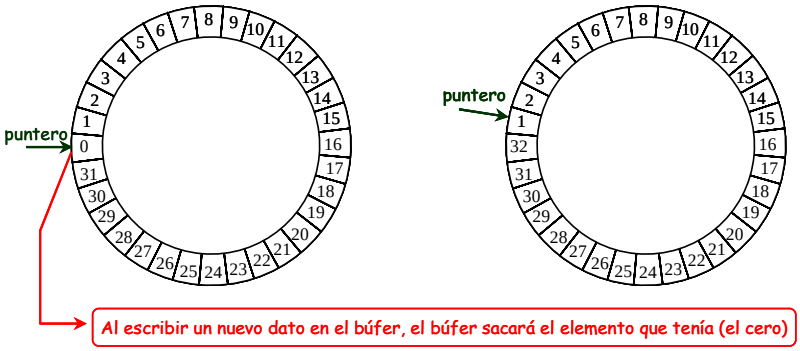


Figura 14.17: Funcionamiento del búfer circular con un único puntero cuando está lleno

El diagrama de bloques simplificado implementado con búferes circulares se muestra en la figura 14.18. Este diagrama es la adaptación con búferes de la versión de la figura 14.9 que tenía registros de desplazamiento. Los tres registros de la derecha de cada fila se mantienen fuera de los búferes para tenerlos disponibles para el procesamiento. Por lo tanto, las búferes tendrán tantos elementos como columnas tenga la imagen menos tres (los tres que se han sacado).

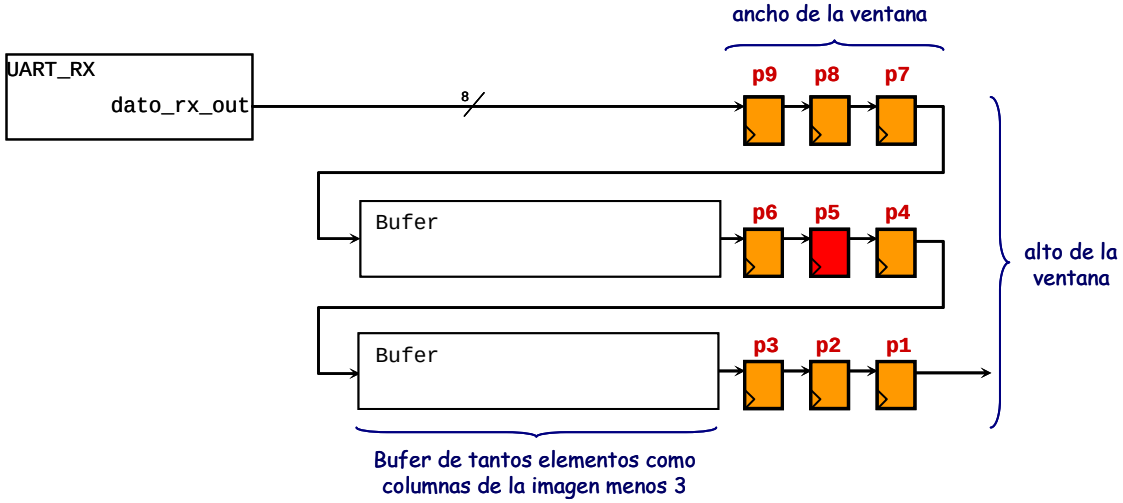


Figura 14.18: Variante del circuito de la figura 14.9 implementado con búferes

El búfer que tenemos que realizar es sencillo ya que cada vez que entra un dato tiene que salir otro (que será el primero que ha entrado). Así que sólo tiene tres puertos (además del reloj y reset), que son: el píxel que entra (`pxl_in`), el píxel que sale (`pxl_out`) y la señal que avisa que va a entrar un nuevo píxel (`metepxl`). La figura 14.19 muestra un posible esquema del búfer. En el circuito hay una RAM de un único puerto y con modo de lectura primero (*read first*). Las memorias RAM de este tipo se vieron en el código 12.12. En el esquema también hay un contador, que es un contador circular que simplemente va aumentando la cuenta cada vez que llega un nuevo píxel. El fin de cuenta coincide con el tamaño de la RAM. Cuando la cuenta llega a su fin, vuelve a empezar desde cero.

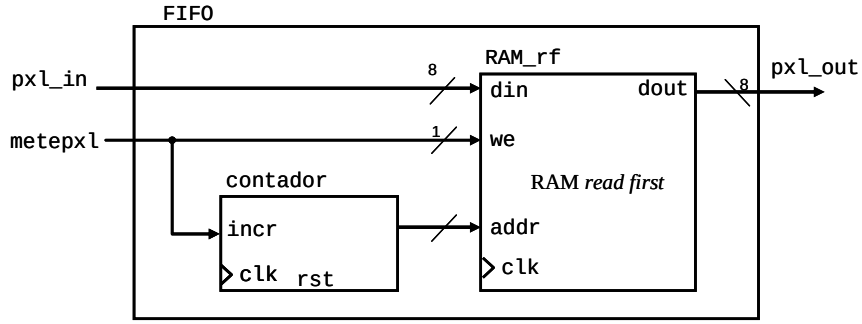


Figura 14.19: Diagrama de bloques de la FIFO

Como siempre, el diagrama de la figura 14.19 es una propuesta que puede variar según como implementes el diseño. Siempre debes comprobar mediante simulación que funciona correctamente.

Si no fuese posible utilizar una memoria con modo de primero lectura, habría que modificar el circuito. Por ejemplo utilizando una memoria RAM con dos puertos, uno para leer y otro para escribir. La cuenta de la lectura estaría adelantada respecto a la de escritura, y la memoria tendría que tener un elemento más. Si no es posible utilizar una memoria de doble puerto, habría que realizar la operación en dos ciclos de reloj, o habría que tener preparado el píxel que se va a leer en un registro auxiliar antes de realizar la escritura. Como ya sabemos, existen muchas posibilidades en el diseño digital y es la tarea del diseñador elegir entre ellas según los requisitos del sistema y el tiempo que tiene para realizar el diseño.

15. Videojuego Pac-Man

Terminaremos este libro haciendo una versión simplificada del videojuego *Pac-Man*, también conocido en España por *comecocos*. En este diseño no haremos nada que no hayamos hecho en los anteriores, pero nos permitirá volver a utilizar y afianzar muchos de conceptos vistos. Al terminar la práctica podrás intentar implementar otros videojuegos clásicos en la FPGA, tal como han realizado los alumnos de DCSE en prácticas: *Super Mario Bros, Space Invaders, Tetris, Arcanoid, Frogger, Zelda...*

Todavía se nos quedarán temas por tratar en este libro, como serían profundizar en la optimización, diseños con varios relojes, uso de memorias externas, microprocesadores, buses, otros tipos de periféricos, ... quizá para otro libro, aún así, con el conocimiento que puedes haber adquirido seguramente tendrás la base suficiente para afrontar diseños más complejos y seguir aprendiendo por ti mismo.

15.1. Videojuego Pac-Man

El videojuego *Pac-Man* se creó en 1980. El juego transcurre en un laberinto, el *pac-man* es una cabeza amarilla con boca que tiene que comer las bolitas que hay por el laberinto sin que sea comido por los fantasmas. Hay cierto tipo de bolitas que son mágicas y permiten al *pac-man* comerse a los fantasmas.

Hay muchas versiones del videojuego, la figura 15.1 muestra la versión de las máquinas recreativas. En la figura 15.2 se muestra la adaptación para Atari creada en 1981.

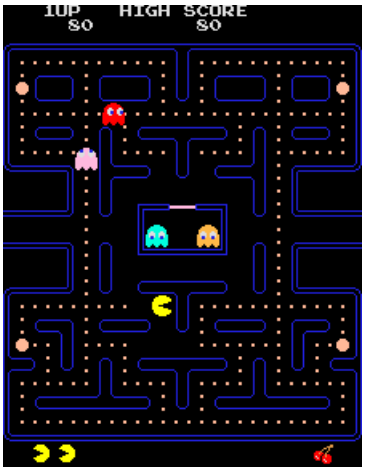


Figura 15.1: Videojuego Pac-Man

Figura 15.2: Versión del Pac-Man para Atari

No pretendemos implementar todas las funcionalidades del videojuego. Haremos una versión más simple que te permitirá, si lo deseas, implementar el videojuego completo por ti mismo.

Como hemos ido haciendo en el libro, implementaremos el diseño paso a paso. Primero crearemos el campo de juego, luego el muñeco del *pac-man* con su movimiento a través del campo de juego. Posteriormente limitaremos el campo de juego con las paredes del laberinto. A continuación crearemos las bolitas que se come el *pac-man*, incluyendo la puntuación. Por último, crearemos a los fantasmas.

Antes de empezar a leer, te animamos a que pienses cómo implementarías el videojuego. Luego, compara lo que has pensado con las propuestas que se van haciendo en este libro, y en caso de que sean diferentes, podrías implementar ambas y comparar cuál te parece mejor. También te recomendamos que hagas lo mismo con cada uno de los apartados. Antes de leer el apartado, dedica un tiempo a pensar tu manera de implementarlo. Un diseño que parece obvio, puede haber requerido mucho tiempo en su elaboración y estructuración, y enfrentarse a esta primera fase es una de las tareas más importantes del diseñador.

15.2. El campo de juego

El campo de juego va a ser una matriz de 32 columnas y 30 filas. En principio, el *pac-man* podrá tener una de estas 960 posiciones (32x30= 960). Cada elemento de la matriz (celda) tendrá 16 píxeles de ancho y 16 de alto[114]. Por lo tanto, el campo de juego tendrá 512 píxeles de ancho y 480 píxeles de alto, estas dimensiones hacen que quepa en nuestra pantalla VGA que tiene 640x480 píxeles. Como el alto del campo de juego mide lo mismo que el de la pantalla, si se quiere incluir un marcador o alguna otra información, como el número de vidas del *pac-man*, se tendrán que poner a los lados. Pondremos el campo de juego a la izquierda de la pantalla y la parte de la información (marcador) a la derecha. La figura 15.3 muestra la matriz del campo de juego y su relación con la pantalla VGA.

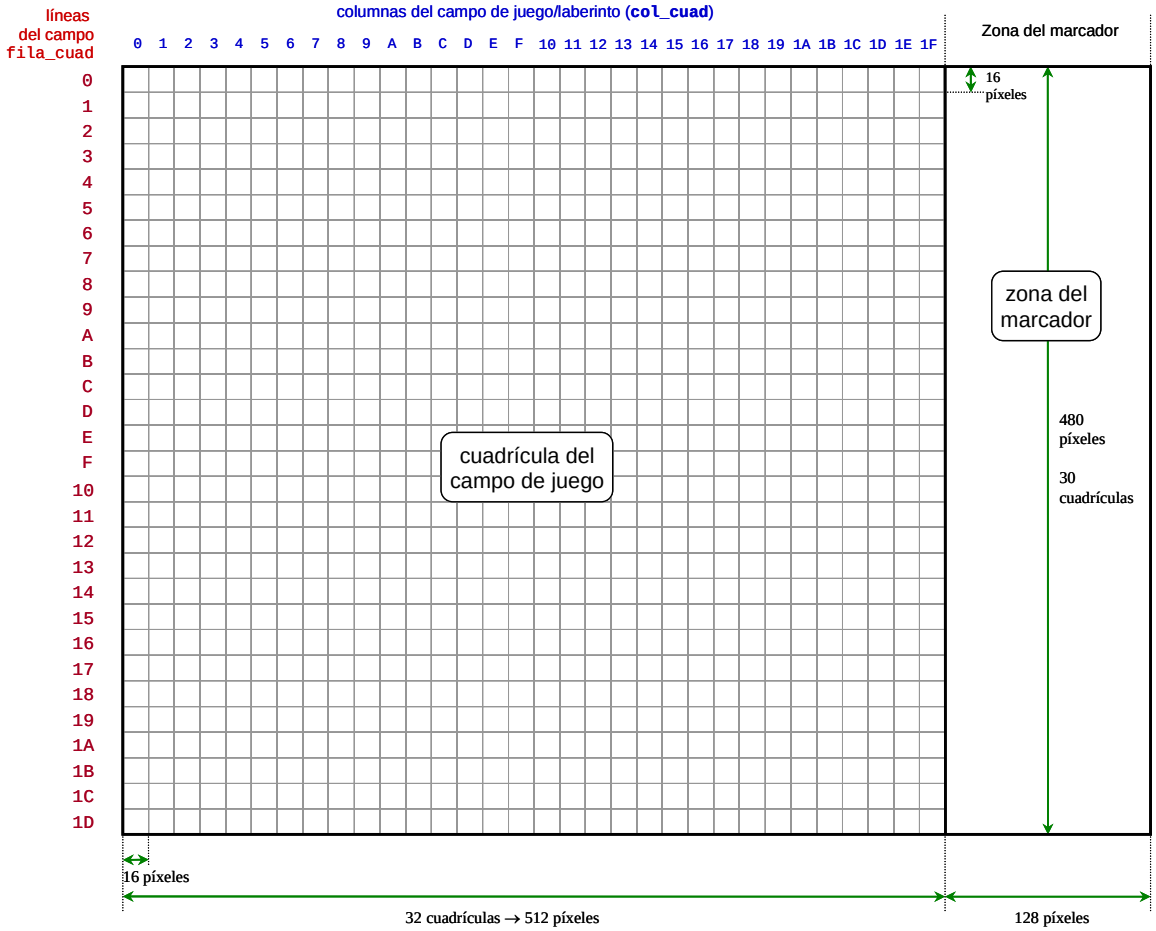


Figura 15.3: Cuadrícula del campo de juego y su relación con la pantalla de 640x480

[114] Con estas dimensiones y la resolución de la pantalla los muñecos salen grandes, puedes optar a reducir las celdas a 8x8 y así incluso poder hacer el laberinto más grande.

Podemos apreciar la similitud con el circuito que dibuja los caracteres por pantalla (capítulo 13 y su figura 13.4). En este caso, las celdas son de 16x16 píxeles. Ya sabes la importancia de elegir números que sean potencias de dos. El cálculo de la columna y fila de la cuadrícula (`col_cuad` y `fila_cuad`) se realiza dividiendo el píxel de la pantalla (`pxl_num`) y la línea de la pantalla (`line_num`) entre 16, esto es lo mismo que quitar los cuatro bits menos significativos. Esto es similar al cálculo realizado en el apartado 13.4, pero en aquel caso se realizaba con cuadrículas de 8x16.

En este caso hemos elegido un campo de juego de 32 columnas para calcular por concatenación el número de la cuadrícula (la celda: `cuad_num`). Puedes elegir otro número distinto y realizar las operaciones realizadas en el apartado 13.4. Sin embargo, si eliges 32 columnas te bastará concatenar adecuadamente `fila_cuad` con `col_cuad`, como se hacía en el apartado 12.2.3.6 (teniendo en cuenta que en ese apartado se concatenaban `line_num` y `pxl_num`).

Antes de seguir, te recomendamos que pruebes la cuadrícula, para ello puedes dibujar un tablero de ajedrez con las cuadrículas, por ejemplo:

- Las filas de la cuadrícula que sean pares, `fila_cuad(0)='0'`, tendrán:
  - las celdas pares (`col_cuad(0)='0'`) en blanco
  - las celdas impares en negro
- Las filas de la cuadrícula que sean impares tendrán:
  - las celdas pares en negro
  - las impares en blanco
- La zona del marcador estará en azul

Una vez que tengas la cuadrícula puedes pasar al siguiente apartado. Recuerda realizar copias de los proyectos y trabajar con las copias, de modo que siempre tengas los proyectos de los apartados anteriores disponibles, por si todo va mal o por si decides retomarlos.

15.3. El jugador sin rostro

Queremos mover el *pac-man* por el tablero de ajedrez (cuadrícula) que hemos hecho. El *pac-man* tendrá unas coordenadas (`col_pacman`¸ `fila_pacman`) referidas a la columna y fila de la cuadrícula (`col_cuad` y `fila_cuad`), por lo que tendrán las mismas dimensiones.

Antes de empezar a diseñar, mejor que crees un proyecto nuevo y copia los archivos del anterior.

Tendremos que añadir al proyecto un módulo de interfaz con los pulsadores[115] para dirigir el movimiento del *pac-man*. Será un módulo similar al explicado en el apartado 10.1, pero ahora el *pac-man* se mueve de celda en celda, por lo tanto, de 16 en 16 píxeles. Si en el interfaz con los pulsadores eliges el modo de movimiento continuo (apartado 10.1), la velocidad de movimiento deberá ser unas 16 veces más lenta que la que hiciste en el diseño del *Pong*. Si el movimiento es por pulsaciones, irá de celda en celda y no habrá problema, aunque quizá sea demasiado lento. Otra opción es que el *pac-man* siempre esté en movimiento, y su dirección venga determinada por el último pulsador que se haya presionado.

[115] También puedes hacer interfaz con el teclado

En cualquier caso, el interfaz con los pulsadores proporcionará las señales de movimiento del *pac-man* en cuatro posibles direcciones: arriba, abajo, derecha e izquierda. Estas señales deberán de ser lo suficientemente lentas para que los movimientos sean visibles y controlables. Sólo una de estas señales podrá estar activa en cada momento, o si no, establecer prioridades entre estas señales.

Estas señales de las direcciones harán cambiar las coordenadas del *pac-man* (`col_pacman`, `fila_pacman`). En el caso de que se superen los bordes del campo de juego, el *pac-man* aparecerá por el otro extremo. Esto es, si está en la primera fila (la de arriba) y tiene la orden de subir, aparecerá por la última fila (la de abajo) sin cambiar de columna. Lo mismo ocurrirá en el otro sentido y en las columnas.

En este diseño, además de mostrar por la VGA el tablero de ajedrez que hiciste en el apartado anterior, deberá marcar en amarillo la celda donde esté el *pac-man*. Recuerda que la celda donde está el *pac-man* viene determinada por `col_cuad` y `fila_cuad`. Estas señales tendrán que ser registros y los tendrás que inicializar mediante el reset al valor que estimes conveniente dentro de la cuadrícula. En la pantalla deberá aparecer la celda del *pac-man* pintada completamente de amarillo. Al presionar los pulsadores la celda amarilla deberá moverse por la pantalla. Comprueba que se mueve a una velocidad adecuada y que al atravesar los bordes aparece en el lado contrario.

15.4. El pac-man

El jugador del diseño anterior no tenía mucha gracia ya que no tenía rostro, queremos mostrar al verdadero *pac-man*. Realmente no hace falta mucha explicación para esta tarea, ya que has hecho algo muy parecido en el capítulo de escribir caracteres por pantalla. En aquel capítulo pintábamos los caracteres correspondientes a las celdas, ahora tenemos que pintar la figura del *pac-man* en una celda determinada.

Tenemos que crear una ROM con la imagen del *pac-man*, puedes escoger el tipo de ROM que prefieras: en blanco y negro, escala de grises o a tres colores. Cómo pintar este tipo de memorias lo vimos en el capítulo 12.

Partiremos del caso que consideramos más sencillo, que es tener la ROM de ancho de palabra igual a los píxeles del ancho de la celda y tantas posiciones de memoria como el alto de la celda. Esto lo hicimos en el código 12.4 y en la escritura de caracteres por pantalla. El código 15.1 muestra la constante que tiene los valores de la ROM, para hacer la ROM simplemente tendrías que copiar esa constante en una arquitectura como la del código 12.4 y verificar que los rangos de las señales son correctos.

```
constant img : memostruct := (
 -- FEDCBA9876543210
   "0000000111100000",-- 0
   "0000011111111000",-- 1
   "0000111111111100",-- 2
   "0001111011111110",-- 3
   "0011111111111100",-- 4
   "0011111111110000",-- 5
   "0111111110000000",-- 6
   "0111111000000000",-- 7
   "0111111000000000",-- 8
   "0111111110000000",-- 9
   "0011111111110000",-- A
   "0011111111111100",-- B
   "0001111111111110",-- C
   "0000111111111100",-- D
   "0000011111111000",-- E
   "0000000111100000" -- F
 );
```

Código 15.1: Constante de la ROM para dibujar el pac-man a dos colores

Así que prueba a implementar el circuito con el muñeco del *pac-man* como protagonista. Si has cogido la memoria del código 15.1, pintarías de amarillo cuando la memoria devuelve un uno y cuando la memoria devuelve un cero, lo mejor sería pintar el fondo. Esto es, cuando hay un cero, no pintar nada en la condición (en el *if* del proceso) que pinta el *pac-man* y dejar que la condición que pinta el fondo ponga el color[116], así parece que el *pac-man* se mueve sobre el fondo. Si no haces esto así y los colores del fondo no coinciden con los del fondo del *pac-man*, parecería que éste llevase una especie de sombra incómoda.

Una vez que lo tengas, comprueba que el muñeco del *pac-man* y no el cuadrado que teníamos antes, se mueve bien por el tablero de ajedrez.

¿En qué sentido te ha salido la boca del *pac-man*? ¿mira a la derecha o a la izquierda? recuerda del apartado 12.2.4 y del capítulo 13 que si no cambiábamos los índices se dibujaba la imagen simétrica. En realidad te debería salir mirando a la derecha, pero el hecho de que se pueda cambiar tan fácilmente es una ventaja. Podemos hacer que mire hacia un lado o hacia el otro según el sentido que tenga el *pac-man*. Sólo habría que guardar en un registro el sentido del último movimiento.

Si quisiésemos hacer que el *pac-man* mire para arriba y para abajo, con la misma figura podríamos hacer una trasposición de filas por columnas. Aunque requiere un poco de análisis y seguramente necesitarás algunas pruebas, ésta es la opción más recomendada. Otra alternativa sería crear otras dos imágenes con el *pac-man* mirando arriba y abajo. Las imágenes se podrían guardar en la misma ROM triplicando el número de direcciones de memoria[117]. La selección de la imagen se haría igual que la selección de los caracteres con la ROM que guardaba los caracteres ASCII.

Una vez hecho todo esto, tendremos al *pac-man* moviéndose por el tablero de ajedrez y mirando en el sentido en el que se mueve.

[116] Por tanto, la condición que pinta el *pac-man* y la que pinta el fondo no deben ser excluyentes, esto es, no deben estar en *if-elsif*, sino que deben ser dos *if* en el que el *pac-man* sobreescribe al fondo.

[117] Crear tres memorias ROM es menos recomendable porque hace aumentar el número de conexiones internas del circuito. Además, nunca se van las distintas imágenes del *pac-man* simultáneamente por lo que no hay que gestionar los accesos independientemente.

15.5. El laberinto

Hasta ahora el *pac-man* se mueve libremente por el tablero de ajedrez. Queremos crear un laberinto y hacer que el *pac-man* vea limitada su libertad de movimiento por las paredes del laberinto.

El laberinto también lo vamos a crear con una imagen en blanco y negro, donde los unos van a ser las paredes del laberinto y los ceros serán los pasillos por donde puede pasar el *pac-man*. La imagen tendrá que tener las mismas dimensiones de la cuadrícula del campo de juego. Pero las dimensiones estarán en cuadrículas y no en píxeles, es decir, la imagen será de 32x30 en vez de 512x480 (recuerda las dimensiones en la figura 15.3). Con esto ahorramos mucha memoria.

Lo que haremos es pintar cada elemento de la imagen del laberinto (cada píxel) en una cuadrícula. Como las cuadrículas son de 16x16, sería como ampliar la imagen por 16. En apartado 12.2.3.5 vimos cómo mostrar una imagen ampliada por dos. Ahora, en vez de quitar el bit menos significativo (ampliación por dos), quitaremos los cuatro bits menos significativos (ampliación por 16). O dicho de otra manera, usaremos las coordenadas de la cuadrícula `col_cuad` y `fila_cuad` para calcular la dirección de memoria y, en su caso, el índice del dato de la memoria.

Así que sólo hace falta crear la imagen del laberinto. En el código 15.2 está la constante de la ROM para el laberinto (puedes crear otro laberinto si lo deseas). La ROM es del mismo tipo que la del *pac-man*: cada fila de la imagen se corresponde con una dirección de memoria, y cada elemento del dato contiene todas las columnas.

```
  constant img : memostruct := (  -- los unos son las paredes
   -- FEDCBA9876543210
     "1111111111111111",-- 0
     "1000000000000001",-- 1
     "1011110111111101",-- 2
     "1011110111111101",-- 3
     "1011110111111101",-- 4
     "1000000000000000",-- 5
     "1011110110111111",-- 6
     "1011110110111111",-- 7
     "1000000110000001",-- 8
     "1111110111111101",-- 9
     "1111110111111101",--10
     "1111110110000000",--11
     "1111110110111100",--12
     "0000000000100000",--13
     "1111110110100000",--14
     "1111110110111111",--15
     "1111110110000000",--16
     "1111110110111111",--17
     "1111110110111111",--18
     "1000000000000001",--19
     "1011110111111101",--20
     "1011110111111101",--21
     "1000110000000000",--22
     "1110110110111111",--23
     "1110110110111111",--24
     "1000000110000001",--25
     "1011111111111101",--26
     "1011111111111101",--27
     "1000000000000000",--28
     "1111111111111111" --29
    );
```

Código 15.2: Constante de la ROM para dibujar el laberinto (mitad)

Si observas la memoria podrás ver que el ancho de palabra de la memoria es de sólo 16 bits, mientras que el campo de juego tiene 32 columnas. Lo hemos hecho así porque el

campo es simétrico, entonces para pintar el lado derecho sólo tenemos que poner la imagen especular del lado izquierdo. Así sólo necesitamos la mitad de la memoria.

Pero antes de pintar la imagen especular, mejor será probar si podemos pintar la mitad de la izquierda[118]. Haz una copia del proyecto anterior que pintaba el *pac-man* e incluye la memoria del código 15.2. Haz que se pinte la imagen de modo que cada píxel de la imagen tenga el tamaño de una cuadrícula. El *pac-man* deberá aparecer por encima de la imagen de fondo. La imagen sólo aparecerá en la mitad izquierda del campo, en las primeras 16 columnas de la cuadrícula (256 píxeles de la VGA).

Cuando lo tengas hecho, ahora vamos repetir la imagen en las siguientes 16 columnas de la cuadrícula (los píxeles del 256 al 511). En el apartado 12.2.3.4 vimos cómo repetir una imagen. Ahora en vez de repetirla en toda la pantalla la limitaremos al campo de juego.

Si simplemente repites la imagen no quedará la imagen simétrica por el lado derecho, tendrás que cambiar los índices de las columnas. Esto lo acabamos de hacer con el *pac-man* para que mirase hacia el lado que camina y lo vimos en el apartado 12.2.4. En realidad lo que tenemos que hacer es deshacer el cambio de índices para que ahora sí salga la imagen especular, pues es lo que queremos. En resumen:

- En la parte izquierda del campo de juego haremos el cambio de índices (apartado 12.2.4). Que son las celdas 0 a la 15, o los píxeles 0 a 255.
- En la parte derecha no haremos el cambio de índices. Celdas 16 a 31 ó píxeles 256 a 511.

No te olvides de elegir colores distintos para las paredes del laberinto y los pasillos.

15.6. Recorrer el laberinto

Tenemos el *pac-man* y el laberinto, pero el *pac-man* cruza las paredes del laberinto sin problema. En este apartado resolveremos esta cuestión.

Para tener en cuenta las paredes del laberinto, cada vez que se recibe una orden de movimiento, el circuito debe de comprobar si la nueva posición está permitida. Si para la nueva posición en la ROM hay un uno indicará que es una pared, y por lo tanto el *pac-man* no se podrá mover en esa dirección. Para hacer la comprobación habrá que acceder a la memoria y comprobar el elemento de la memoria correspondiente a la posición deseada. Como también se accede a la ROM para pintar el campo de juego, tendríamos que realizar un control de los accesos, ya que queremos acceder a ella desde dos sitios diferentes: para pintar y para comprobar la nueva posición. Realmente no es mucho problema ya que los movimientos del jugador son muy lentos comparados con las frecuencias de la placa. Sin embargo, para ahorrarnos ese control, podemos hacer que la memoria ROM del campo de juego sea de doble puerto y acceder a ella de forma independiente[119].

Para hacer la comprobación es conveniente que hagas una máquina de estados, ya que los accesos a memoria no son instantáneos ni tampoco la actualización del registro que tendrás que escribir para indicar a la memoria la dirección que quieres leer. Por último, si la celda es accesible, tendrás que actualizar las nuevas coordenadas del *pac-man*.

[118] Si quieres, también puedes hacer la imagen completa y no recurrir a estos trucos para dibujar la imagen especular

[119] Realmente estas memorias son muy pequeñas e incluso se pueden definir como constantes en el código VHDL. Esto simplificaría enormemente el circuito, pero lo estamos haciendo así para familiarizarnos con el uso de las memorias.

Es importante también que la posición inicial del *pac-man* sea una posición permitida (sea un cero en la ROM del laberinto).

Implementa el circuito y comprueba que el *pac-man* sólo se mueve por los pasillos, comprueba también que entra por el pasadizo que hay en los laterales y que sale por el lado opuesto.

15.7. Control del movimiento del pac-man *

El movimiento del *pac-man* no es muy suave. Si escogemos el movimiento por pulsación, es decir, cada vez que presionamos el pulsador se mueve una celda, el movimiento es muy lento. Por otro lado, si escogemos el movimiento continuo a veces puede ser difícil desviar el rumbo por pasillos laterales. Podemos implementar un movimiento más sofisticado que haga lo siguiente:

- Al presionar un pulsador el *pac-man* sigue la dirección fijada de manera indefinida hasta que encuentre una pared que le impida continuar. Es decir, no hace falta mantenerlo pulsado para que continúe el movimiento.
- Si el *pac-man* se mueve en una dirección y presionamos un pulsador de otra dirección, el *pac-man* continuará moviéndose en la dirección inicial hasta que encuentre un pasillo que le permita cambiar a la nueva dirección.

Esta manera de moverse resulta más fácil para controlar el movimiento del *pac-man* y poder meterse por pasillos laterales.

Para implementar este movimiento necesitarás crear una máquina de estados y un temporizador que genere pulsos para ordenar el desplazamiento periódico del *pac-man*. El temporizador determinará la velocidad del *pac-man*. Otra alternativa es permitir presionar dos pulsadores simultáneamente siempre que no sean de direcciones opuestas. Sería similar a un mando (*joystick*) que permite direcciones oblicuas.

La implementación de este control de movimiento no es necesaria, pero mejora enormemente el control del *pac-man*.

15.8. La comida y los puntos

El *pac-man* tiene que comerse todas las bolitas de comida que hay en el laberinto y esto le permite pasar de nivel. Cada bolita de comida le da un punto. Para implementar la comida puedes crear una memoria RAM del tamaño de la cuadrícula (32x30). Esta memoria RAM tendrá todos sus valores a cero (hay comida) y cada vez que el *pac-man* pase por una celda, se escribirá un uno en la posición de memoria correspondiente. Por lo tanto, un cero en la memoria RAM significaría que hay comida o lo que es lo mismo, que el *pac-man* no ha pasado por allí. Y un uno significará que no hay comida o que el *pac-man* ha pasado.

Si el *pac-man* pasa dos veces por la misma celda se puede volver a escribir un uno en la memoria, no pasa nada por escribir de nuevo un uno en la memoria cuando ya había un uno. Lo que sí es importante es no volver a sumar puntos. Tendrás que crear un contador de puntos que se vaya incrementando cuando el *pac-man* entre en una celda con comida (con un cero en la memoria RAM de la comida).

Si se hace de esta manera habrá puntos que nunca se podrán comer, pues la memoria RAM de la comida contiene todas las celdas de la cuadrícula, incluyendo las celdas de pared del laberinto por donde el *pac-man* nunca podrá transitar. No pasa nada, lo importante es tener en cuenta el número máximo de celdas transitables de un laberinto. Tendremos que crear un contador para cada laberinto que cuente el número de bolitas que se han comido (celdas transitadas), cuando se llegue al número máximo de celdas transitables se habrá terminado la pantalla. Este número máximo de celdas transitables lo podrás calcular a mano y ponerlo como constante para cada laberinto, o también puedes crear un circuito que en la inicialización cuente las celdas transitables del laberinto. La primera opción es más fácil de hacer para una pantalla, mientras que la segunda opción es más generalizable y produce menos errores.

De manera similar a la memoria ROM del laberinto, se accede a la memoria RAM de la comida desde dos sitios diferentes. Por un lado se accede al dibujar la pantalla y por otro lado se accede para contabilizar los puntos y escribir cuándo se transita por las celdas. Por lo tanto podemos implementar la memoria RAM con doble puerto para independizar estas dos partes.

Cuando se pinta el laberinto en la pantalla, entre otras, hay que hacer las siguientes comprobaciones:

- Si es pared: se pinta la pared como se ha hecho hasta ahora. No se mira si hay comida, pues en la pared no se pinta la comida.
- Si es pasillo:
  - Si no hay comida: se pinta el pasillo como hemos hecho hasta ahora
  - Si hay comida: se pinta una bolita de comida con el mismo color de fondo que el pasillo. Para la bolita de comida puedes crear una imagen de la misma manera que el *pac-man*, también puedes crear imágenes más sofisticadas con más colores, o simplemente pintar de un color distinto los píxeles centrales de la celda.

Por último, puedes incluir un marcador a la derecha que indique el número de puntos del *pac-man*, que por ahora es la cuenta del número de bolitas que se ha comido.

15.9. Las paredes del laberinto *

Hemos pintado las paredes del laberinto de un color, coloreando toda la cuadrícula del mismo color. Haciéndolo así, las paredes se parecen más a la versión de Atari (figura 15.2) que a la versión de las máquinas recreativas (figura 15.1). Podemos dejar las paredes como están, pero si queremos un acabado más similar al de las máquinas recreativas podemos "pintar" las paredes.

En la figura 15.4 se muestran las paredes como están ahora. Para no dibujar el campo de juego entero, sólo se muestra la esquina superior izquierda. En la figura se ha escogido el negro como pared y el blanco como pasillo, pero cualquier otra combinación de colores es válida. En la figura 15.5 se muestra el diseño cuadriculado que dibujaría bordes redondeados. Este diseño sería similar al de la versión de las máquinas recreativas (figura 15.1), aunque en el dibujo se ha escogido una combinación de colores distinta para facilitar la visualización.

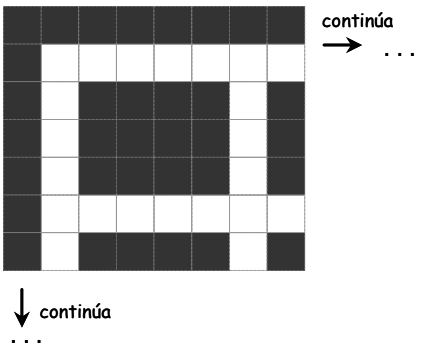


Figura 15.4: Paredes rellenas con un sólo color

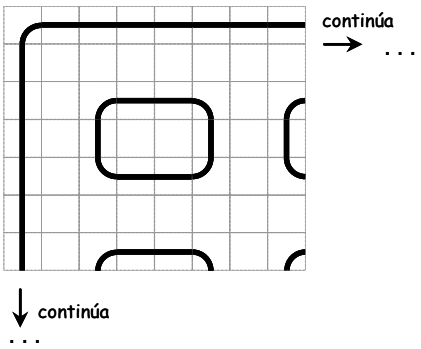


Figura 15.5: Paredes rellenas con dibujos

Observando la figura 15.5 podemos ver que hay siete tipos de celdas para pintar la pared. Estas celdas se muestran en la figura 15.6. Recordemos que en nuestro diseño las celdas son de 16x16, por lo tanto, las formas circulares aparecerán un poco pixeladas, igual que lo era la imagen del *pac-man*. El ancho de la línea puede ser de dos píxeles y estarán centrados en la cuadrícula.

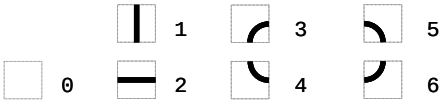


Figura 15.6: Tipos de celdas para pintar las paredes del laberinto

Por tanto se podría crear una memoria ROM que tuviese las imágenes de los tipos de celda de las paredes del laberinto. Los tipos de celda a pintar se pueden reducir a dos, ya que el tipo cero (figura 15.6) no necesita memoria; los tipos uno y dos, son el mismo dibujo cambiando filas por columnas; y los tipos tres, cuatro, cinco y seis, son el resultado de distintos cambios de filas y columnas (similar a las del dibujo del *pac-man* visto en el apartado 15.4). Puedes elegir lo que te sea más fácil: incluir en la ROM dos tipos de celdas y hacer cambios de filas y columnas, o incluir seis tipos de celdas.

El tipo de celda de la pared viene determinada por las cuatro celdas adyacentes (arriba, derecha, abajo e izquierda). Dependiendo de si las celdas adyacentes son de pasillo o de pared, el tipo de celda a dibujar en la pared será distinto. También hay que considerar como casos especiales los bordes del campo de juego. El cálculo para obtener del tipo de pared que hay que pintar es sencillo, la figura 15.7 muestra cuatro de las 16 posibles combinaciones[120] (sin contar los bordes del campo de juego). Hay un caso que no se ha considerado, que es la pared rodeada de pasillo en las cuatro direcciones y su implementación podría ser un círculo.

[120] Sin contar los bordes del laberinto. Además, estos en el juego original (figura 15.1) tienen una línea doble, igual que el cuarto central de donde salen los fantasmas.

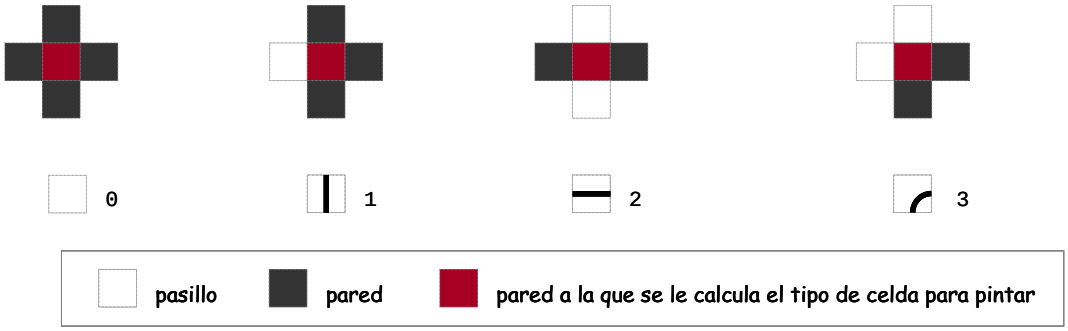


Figura 15.7: Obtención del tipo de celda para pintar las paredes a partir de las celdas adyacentes

Para saber qué tipo de pared hay que pintar en cada caso hay varias alternativas:

- Usar otra memoria ROM que nos indique para cada celda de pared qué tipo es. Los valores de esta memoria los calcularíamos manualmente[121] según los valores de la memoria ROM del laberinto (apartado 15.5). Como el número de tipos de pared son más de dos, la memoria necesitaría más de un bit por celda, y por tanto, la memoria deberá de tener tantas posiciones de memoria como celdas del laberinto. Es decir, no se puede poner toda una fila en una posición de memoria como hemos estado haciendo con las memorias del *pac-man* y del laberinto.

  Por otro lado, recuerda que la memoria del laberinto estaba a la mitad, porque la parte derecha es simétrica a la izquierda. Esto se puede seguir haciendo para esta nueva memoria ROM sabiendo que a la hora de pintar las paredes de la izquierda que sean del tipo 3 y 4 en la parte derecha se cambiarán por las del tipo 5 y 6 respectivamente y a la inversa (las del 5 y 6 se cambiarán por las del 3 y 4). Los otros tipos de paredes son simétricos.

  Las celdas que son de pasillo se pueden dejar como de tipo cero (figura 15.6). Lo peor de este método es que requiere hacer a mano la ROM cada vez que se crea un nuevo laberinto.
- Otra alternativa similar a la anterior es crear una única memoria ROM para el laberinto y el tipo de pared. Bastaría simplemente con incluir un tipo de celda más que fuese "pasillo". Con esto tendríamos ocho tipos de celdas (recuerda la figura 15.6), que justamente cabrían en un byte.
- Se puede realizar un módulo al empezar que calculase el tipo que se pintará para cada celda de pared. Se calcularía a partir de la ROM del pasillo y se guardaría en una memoria RAM. Una vez guardados los datos en la memoria RAM este módulo quedaría inactivo y la memoria RAM funcionaría de manera similar la ROM de las alternativas anteriores. La gran ventaja de este método es que si se implementan varios niveles con distintos laberintos, el circuito se encarga de realizar los cálculos, sin tener que realizarlos manualmente. Por otro lado, en los casos anteriores se necesita una ROM con el tipo de celda de pared para cada laberinto, en este caso sólo se necesita una RAM, en la que se guardan los datos de las celdas al principio de cada laberinto. El tiempo empleado para realizar el cálculo es insignificante y el jugador no se daría cuenta. Este módulo tendría muchas características similares a los módulos de procesamiento de imágenes que realizamos en el apartado 14.2.

[121] También se podría crear un ejecutable que genere las memorias a partir de una imagen, similar al descrito en el apartado 12.2.6

- Por último, también se podría calcular el tipo de celda de pared en el mismo momento que se está pintando. La dificultad de este método es que para el cálculo se necesitan cinco píxeles de la memoria del laberinto (código 15.2), que son el píxel de la celda que se está pintando más los cuatro píxeles adyacentes. Esto hace que se requieren varios accesos a memoria, por lo que hay que anticiparse para que los retardos no hagan que no se muestre el píxel correcto. La ventaja de esta implementación es que no se necesita una memoria para guardar los tipos de imágenes a pintar. Este módulo tendría algunas similitudes con el procesamiento al vuelo del apartado 14.4.2, pero ahora resulta más sencillo porque cada fila está en una posición de memoria, y en el apartado 14.4.2 teníamos un píxel en cada posición de memoria.

Para terminar, se pueden crear paredes del laberinto más sofisticadas con muy poco coste, ya que sólo se necesitan unos pocos tipos de celdas. En vez de los sencillos dibujos de la figura 15.6 se pueden crear celdas con tres colores y más profundidad de color.

15.10. El tamaño del pac-man *

Si has implementado la propuesta del apartado anterior, el *pac-man* quedará pequeño frente al ancho del pasillo. Esto es porque las paredes del pasillo no ocupan toda la celda sino la zona central. En la figura 15.8 se puede comparar el tamaño de los pasillos con la figura del *pac-man*.

Figura 15.8: Tamaño del pac-man comparado con el antiguo y el nuevo pasillo

Así que opcionalmente se puede ampliar el tamaño del *pac-man* al doble del tamaño original para que ocupe la mayor parte del pasillo. La figura 15.9 muestra cómo quedaría el *pac-man* en relación con el nuevo pasillo.

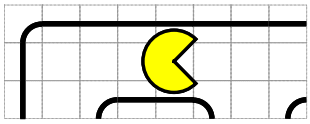


Figura 15.9: Pac-man ampliado para que ocupe la mayor parte del pasillo

La implementación de esta variante es fácil. Hay que crear una memoria ROM para el *pac-man* que tenga el doble del tamaño del anterior, en nuestro caso de 32x32. Podríamos también utilizar la que teníamos, mostrándola al doble de tamaño. Sin embargo, como el *pac-man* es redondo, quedará mejor si lo dibujamos a 32x32 de lo contario quedará muy pixelado.

Por último sólo queda pintarlo. En el proceso que asigna los colores de la VGA ya no podemos seleccionar el área del *pac-man* basándonos en la fila y columna de la cuadrícula (`fila_cuad=fila_pacman` y `col_cuad=col_pacman`) ya que ahora el *pacman* ocuparía también parte de las ocho cuadrículas adyacentes, tal como se puede ver en la figura 15.10. Por lo tanto, ahora tenemos que calcular los píxeles y líneas de la pantalla donde empezaría a dibujarse la imagen (referidos a la pantalla y no a la cuadrícula).

La figura 15.10 muestra en qué píxel de la pantalla hay que empezar a mostrar el *pac-man*, simplemente hay que restar 8 (la mitad de una celda) a la columna y a la fila de la VGA

donde comienza la cuadrícula del *pac-man*. La columna y la fila del *pac-man* (`col_pacman` y `fila_pacman`) se multiplican por 16 porque éstas son la columna y fila de la cuadrícula, y queremos obtener la columna y la fila de la VGA.

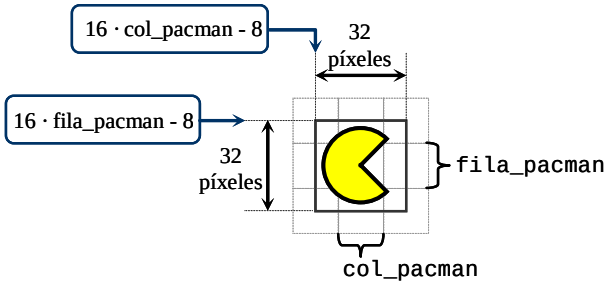


Figura 15.10: Coordenadas del pac-man ampliado

Ahora sólo queda dibujar una imagen de 32x32 a partir del píxel de la pantalla que acabamos de calcular. Esto ya lo hemos hecho varias veces (recuerda el apartado 12.2.3.3).

15.11. Movimiento continuo del pac-man *

Nuestro *pac-man* se mueve dando saltos de celda en celda, es decir, de 16 en 16 píxeles. Esto no es un gran problema, pero es más agradable verlo mover de un modo más continuo. Para ello podemos implementar un movimiento que vaya de píxel a píxel. Aunque el *pac-man* se mueva de píxel en píxel, esto sólo será una apariencia ya que realmente el *pac-man* se seguirá moviendo de cuadrícula en cuadrícula. Es decir, las posiciones del *pac-man* seguirán estando referidas a la cuadrícula (`col_pacman` y `fila_pacman`).

Este movimiento del *pac-man* sólo tiene sentido si hemos implementado un movimiento continuo del *pac-man*. Si el movimiento del *pac-man* viene determinado por pulsaciones individuales de los pulsadores, no tiene mucho sentido, aunque también se podría implementar y seguramente quedaría mejor.

Si el movimiento no está generado por las pulsaciones de los pulsadores, el *pac-man* tiene una velocidad de movimiento determinado por un contador que, o bien muestrea los pulsadores (apartado 15.3) o bien genera pulsos que ordenan el movimiento del *pac-man* (apartado 15.7).

Lo que habría que hacer es establecer otra cuenta 16 veces menor que ordene el movimiento aparente del *pac-man* de píxel en píxel. Habrá muchas maneras de implementar este control, por ejemplo, se podría hacer una cuenta nueva que sea 16 veces menor que la cuenta original, y ahora, la cuenta original, en vez de contar ciclos de reloj, contará 16 pulsos de la nueva cuenta. Llamaremos a esta cuenta `pacm_pxlmov` y haremos que esa cuenta sea descendente, de quince a cero. La figura 15.11 muestra de manera esquemática el movimiento.

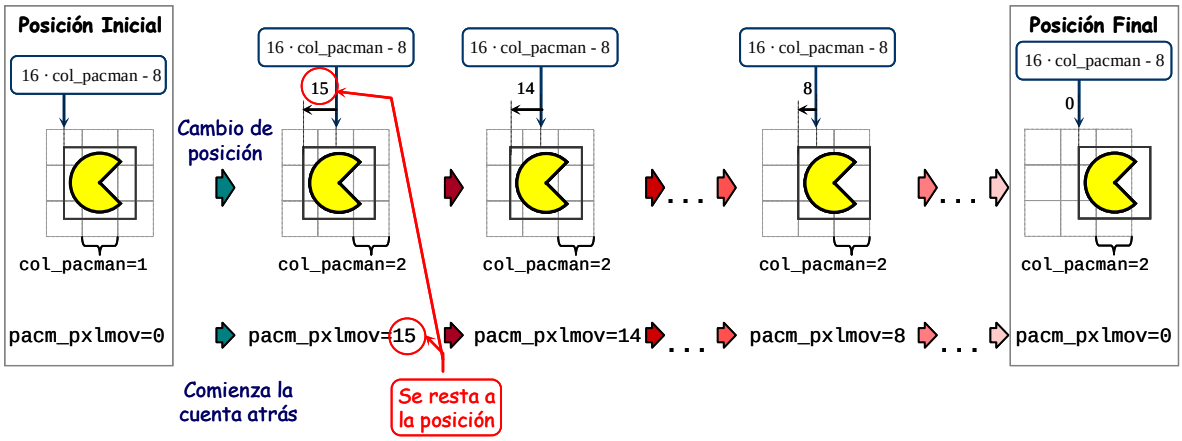


Figura 15.11: Movimiento del pac-man píxel a píxel hasta llegar a la posición final

Analicemos el movimiento del *pac-man*:

- El *pac-man* parte de la posición inicial estando en la columna uno de la cuadrícula (`col_pacman=1`).
- El *pac-man* recibe la orden de moverse a la derecha, por lo tanto pasa a estar en la columna dos (`col_pacman=2`).
- En este momento comienza la cuenta atrás de `pacm_pxlmov` y vale quince. El valor de `pacm_pxlmov` se puede entender como el número de píxeles que le quedan para llegar a su destino.
- Para pintar el *pac-man* se calculan sus coordenadas como se hizo en la figura 15.10 pero al valor de la columna se le resta el valor de `pacm_pxlmov`.
- Se continúa haciendo lo mismo, y el *pac-man* se irá acercando a su destino píxel a píxel, hasta que `pacm_pxlmov` sea cero, y entonces el *pac-man* ya dibujará centrado es su celda.

En el ejemplo de la figura 15.11 el *pac-man* se está moviendo a la derecha, por lo tanto `pacm_pxlmov` se resta a la columna. En el caso de que el *pac-man* se estuviese moviendo a la izquierda, el valor de `pacm_pxlmov` se sumaría al valor de la columna. Y en los casos en los que el movimiento fuese hacia arriba o hacia abajo, el valor de `pacm_pxlmov` se sumaría o restaría a las coordenadas de la fila.

15.12. El pac-man comiendo *

Nuestro *pac-man* va con la boca abierta por todas partes, podemos hacer que abra y cierre la boca mientras camina y se come las bolitas de comida. Incluir esto también es sencillo, sobre todo si has implementado el movimiento continuo del apartado anterior. Solamente tenemos que incluir en la ROM una imagen del *pacman* con la boca cerrada (un círculo completo) que al combinarla con la de la boca abierta parecerá que abre y cierra la boca.

Si se desea un mayor realismo se pueden crear varias imágenes con la boca más o menos abierta e irlas combinando durante el movimiento. La cuenta `pacm_pxlmov` del apartado anterior viene muy bien para la implementación de este movimiento de la boca. Según el valor de la cuenta se seleccionarán unas u otras imágenes del *pac-man* con la boca más o menos abierta. El cambio de las imágenes dependerá de la velocidad de movimiento del *pac-man* y del número de imágenes que se tengan. Cuanto mayor número de imágenes, más similares serán las imágenes contiguas y por lo tanto más rápidamente se podrán

cambiar. Pero hay que tener en cuenta que cambios muy rápidos no son perceptibles por el ojo/cerebro humano.

15.13. Los fantasmas

Llega el momento de incluir a los malos del juego. Para los fantasmas tendrás que crear imágenes de ellos, si has ampliado el *pac-man* como se sugería en el apartado 15.10, tendrás que crear los fantasmas del mismo tamaño. No hace falta crear una imagen para cada fantasma, basta con usar la misma imagen y pintarla de distinto color. Eso sí, cada fantasma deberá tener sus propias coordenadas.

Los ojos de los fantasmas miran hacia donde caminan, puedes implementar esto usando cuatro imágenes distintas con cuatro direcciones de ojos, o mejor aún, puedes crear imágenes de los ojos independientes. Estos ojos se superpondrán a los fantasmas y así también podremos tener los ojos sin fantasma, que es lo que ocurre cuando son comidos por el *pac-man* y tienen que volver a su casa. Esto será en caso de que implementes las bolitas de comida mágicas. En caso de que superpongas los ojos tendrás que definir un color transparente para que se pinte el fondo (el fantasma).

Además no hace falta crear imágenes de ojos de 32x32[122], sino que como los ojos siempre estarán en la mitad superior, podemos olvidarnos de la mitad inferior. E incluso, como los ojos son iguales, podemos guardar un sólo ojo y repetirlo en la otra mitad. Así que la imagen de los ojos puede ser de la mitad: 16x16, si los fantasmas son de 32x32.

Y no sólo esto, sino que como los ojos que miran a la derecha son simétricos con los ojos que miran a la izquierda (simetría vertical); e igual con los que miran arriba y abajo (simetría horizontal), nos podría bastar con dos imágenes de un sólo ojo.

De la misma manera que el *pac-man*, los fantasmas también tendrán unas coordenadas basadas en la cuadrícula pero los fantasmas podrán simular un movimiento continuo similar al del *pac-man* que vimos en el apartado 15.11.

El movimiento de los fantasmas no está dirigido por los pulsadores, así que tenemos que crear nosotros el movimiento. Este movimiento debe ser más o menos aleatorio. El movimiento no debe ser muy aleatorio porque si no sería muy errático, los fantasmas deben tener una cierta continuidad en el movimiento. Para introducir aleatoriedad puedes basar la toma de decisiones en sucesos externos, por ejemplo, el número de ciclos de reloj que han pasado desde la última vez que se ha presionado/soltado un pulsador. O por ejemplo, basarlo en el número de celda donde está el *pac-man*. Por ejemplo, tomando los últimos bits de la fila y columna de la celda, y basando la nueva dirección en los valores de estos bits. También puedes dotarlos de cierta inteligencia para que intenten encontrar al *pac-man*.

Por último, hay que controlar si el fantasma mata al *pac-man*, y las vidas del *pac-man*.

15.14. Otras mejoras *

Si has implementado todas las propuestas es más que suficiente. Pero igual que sucede con casi todo programa informático, siempre hay algo que se puede mejorar en un circuito

[122] O imágenes de 16x16 en el caso de que los mantengas del tamaño de la cuadrícula

digital. Llegados a este punto creemos que, si lo deseas, puedes implementar por ti mismo el resto de características del juego.

Puedes incluir:

- Las cuatro bolitas de comida mágicas que hacen que el *pac-man* tenga poderes durante un tiempo y pueda comerse a los fantasmas. Incluir parpadeo, cambio de color e incluir boca en los fantasmas durante el tiempo en el que son vulnerables. E incluso, combinar el color normal con el color vulnerable para avisar de que queda poco tiempo de su vulnerabilidad
- Incluir el movimiento de las faldas de los fantasmas
- Hacer que los fantasmas vuelvan a casa cuando son comidos
- Controlar la puerta de la casa de los fantasmas y evitar que el *pac-man* pueda entrar. No incluir comida en la casa de los fantasmas
- Incluir los premios que aparecen en las pantallas: cerezas, fresas, naranjas,... y asignar puntos cuando se los coma el *pac-man*
- Incluir varias pantallas y niveles, con diferentes laberintos
- Añadir vidas cuando se supera cierta puntuación
- Escenificar la muerte del *pac-man*
- Incluir pantallas iniciales y finales
- Memoria con la máxima puntuación conseguida
- Hacer que el *pac-man* vaya más despacio cuando está comiendo
- Implementar el control del movimiento desde el teclado y no desde los pulsadores

Y todo lo que veas que falta y te gustaría incluir, es tu juego

15.15. Conclusiones

En el diseño de este videojuego hemos visto que cada característica podía implementarse de muchas maneras. Algunas son claramente más eficientes, pero en otras nos podemos plantear si vale la pena el ahorro considerando los recursos que tenemos. Por ejemplo, si tenemos en cuenta los recursos de la FPGA y los pequeños tamaños de las memorias de los personajes (*pac-man*, fantasmas,...) los ahorros son insignificantes. Sin embargo no siempre es así, y por ejemplo en otros diseños, las memorias son un factor limitante. Recuerda por ejemplo la tabla 12.3 en donde se mostraban las limitaciones para usar imágenes grandes.

Así que estas cuestiones siempre tienen que estar presentes en la mente del diseñador: mejorar las prestaciones y valorar si merece la pena el ahorro conseguido frente al esfuerzo de implementar dichas mejoras.

Para terminar, como dijimos al comienzo de este capítulo, queda mucho por aprender y de hecho pensamos que el diseñador nunca deja de aprender, pero esperamos que lo visto en este libro te valga para que puedas continuar el aprendizaje por ti mismo. Esperamos también que te hayas divertido mientras has estado aprendiendo o que al menos que haya sido entretenido.

Referencias

[1adept] *Adept* de *Digilent*. Programa gratuito para programar las FPGAs por USB: http://www.digilentinc.com/Products/Detail.cfm?Prod=ADEPT

[2ash] P. Ashenden, "Digital Design, An Embedded Systems Approach Using VHDL", Ed. Morgan Kaufmann, 2008

[3basys] *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
Página web de la tarjeta:
http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,791&Prod=BASYS

[4chu] P. Chu, "FPGA Prototyping Using VHDL Examples", Willey, 2008

[5cc] Creative Commons. http://creativecommons.org/
Licencia de este manual: http://creativecommons.org/licenses/by-nc-nd/3.0/

[6ceng] Computer-Engineering: http://www.computer-engineering.org/ps2protocol/

[7desch] J-P. Deschamps, "Síntesis de circuitos digitales, un enfoque algorítmico", Thompson 2002.

[8digi] Digilent Inc. http://www.digilentinc.com/

[9dte] Departamento de Tecnología Electrónica, Universidad Rey Juan Carlos. http://gtebim.es

[10dcse] Diseño de Circuitos y Sistemas Electrónicos. http://gtebim.es/docencia/DCSE

[11ed2] Electrónica Digital II. http://gtebim.es/docencia/EDII

[12educ] 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

[13ieee] IEEE, *Institute of Electrical and Electronics Engineers*. http://www.ieee.org

[14imag] ImageMagik: http://www.imagemagick.org

[15irfan] IrfanView: http://www.irfanview.com

[16ise] ISE WebPack de Xilinx. http://www.xilinx.com/tools/webpack.htm
Para descargar versiones antiguas:
http://www.xilinx.com/webpack/classics/wpclassic/index.htm

[17mach] F. Machado, S. Borromeo, "Diseño de circuitos digitales con VHDL", 2010. Libro Electrónico disponible en el archivo abierto de la Universidad Rey Juan Carlos. http://hdl.handle.net/10115/4045

[18mach] F. Machado, S. Borromeo, N. Malpica, "Diseño digital avanzado con VHDL ", Ed. Dykinson, 2009.

[19mach] F. Machado, S. Borromeo, N. Malpica, "Diseño digital con esquemáticos y FPGA", Ed. Dykinson, 2009.

[20model] Modelsim, http://www.model.com

[21nexys] *Nexys2*, tarjeta con FPGA fabricada por *Digilent*. Manual de referencia:
http://www.digilentinc.com/Data/Products/NEXYS2/Nexys2_rm.pdf
Página web de la tarjeta:
http://www.digilentinc.com/Products/Detail.cfm?Prod=NEXYS2

[22openc] Opencores. http://www.opencores.org

[23qihw] Qi-hardware: http://en.qi-hardware.com/wiki/Teclado

[24realt] RealTerm. http://realterm.sourceforge.net/

[25rtl] IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, IEEE Std 1076.6

[26spart3] *Spartan-3E FPGA Family: Data Sheet*. DS312, Xilinx

[27urjc] Universidad Rey Juan Carlos, http://www.urjc.es
[28web] Página web con código VHDL complementario a este libro: http://gtebim.es/~fmachado/electronica
[29xilinx] Xilinx, http://www.xilinx.com
[30xst] XST User Guide 9.2i. Xilinx, http://ww.xilinx.com/itp/xilinx92/books/docs/xst/xst.pdf
[31xup] *Xilinx University Program Virtex-II Pro Development System. Hardware reference manual*. UG069 v1.0 9 marzo 2005. http://www.xilinx.com/univ/xupv2p.html