MÁSTER OFICIAL EN INGENIERÍA ELECTRÓNICA

SISTEMAS INTEGRADOS

BLOQUE 4: Microprocesador MicroBlaze de Xilinx

Curso 2013-2014

José Torres

Raimundo García

Julio Martos

Jesús Soret

Adrián Suárez

Pedro A. Martínez

Abraham Menéndez

VNIVERSITAT DE VALÈNCIA Escola Tècnica Superior d'Enginyeria

SISTEMAS INTEGRADOS

Tema 4.- Microprocesador MicroBlaze de Xilinx

Máster Oficial en Ingeniería Electrónica
Curso 2013-14

VNIVERSITAT DE VALÈNCIA Escola Tècnica Superior d'Enginyeria

Microprocesador MicroBlaze de Xilinx

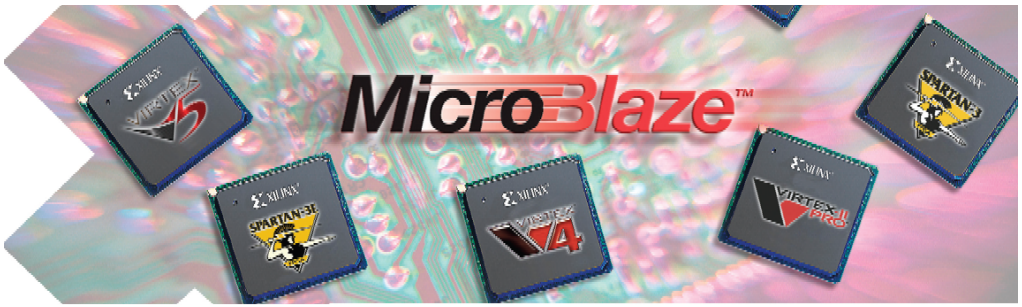
Índice

- Introducción a Microblaze.
- Buses e interfaces.
- Periféricos de usuario

Microprocesador MicroBlaze de Xilinx

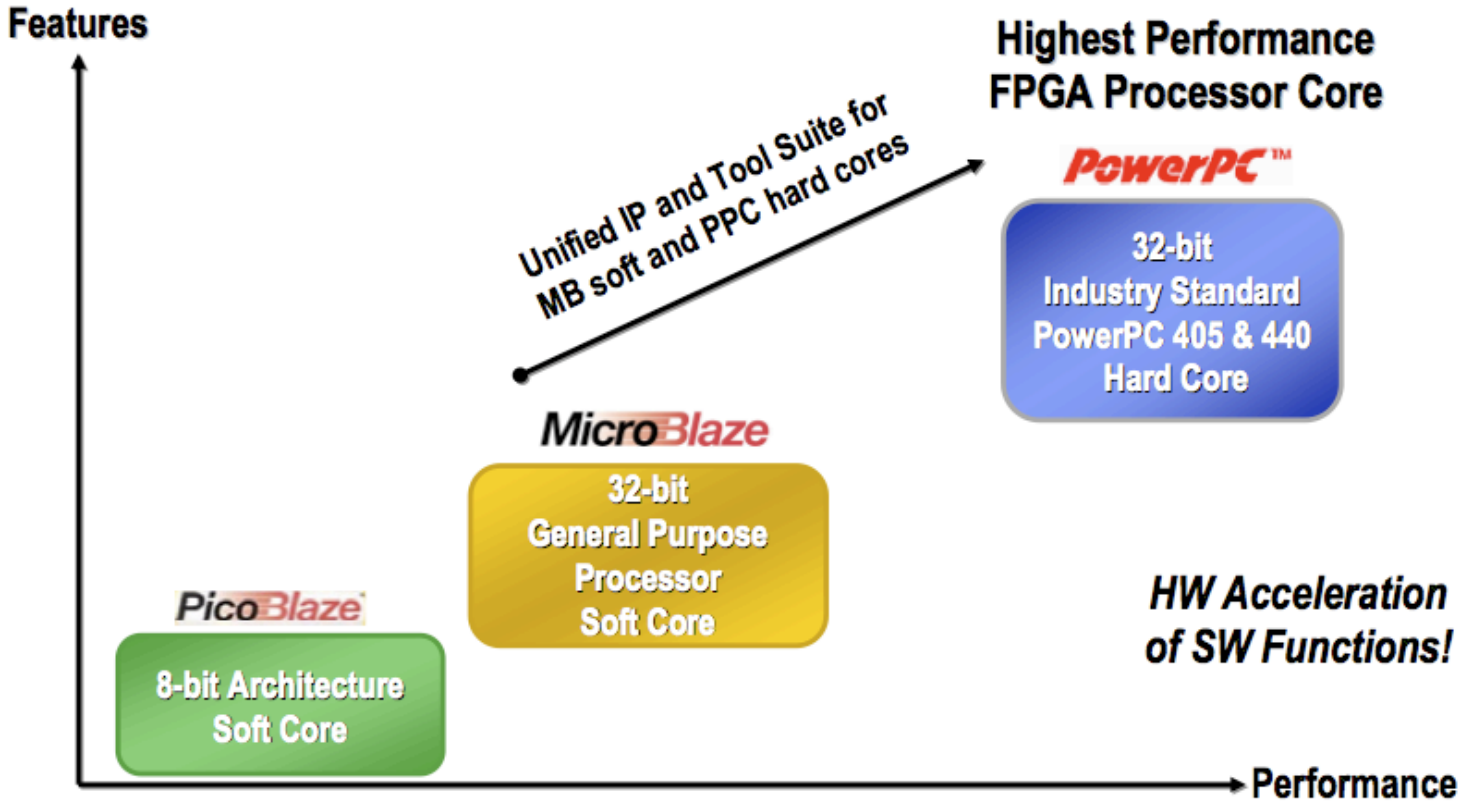
Introducción MicroBlaze

- MicroBlaze es un bloque microprocesador “software” diseñado para su implementación en FPGAs de Xilinx.
- Presenta un juego de instrucciones reducido (RISC)
- Acceso a instrucciones y datos de forma independiente (Harvard)
  - Es habitual que estén en la misma memoria física para facilitar la depuración software.
- La última versión de MicroBlaze que incorpora EDK 10.1 es la 7. Actualmente estamos en la versión 8.4



Introducción MicroBlaze

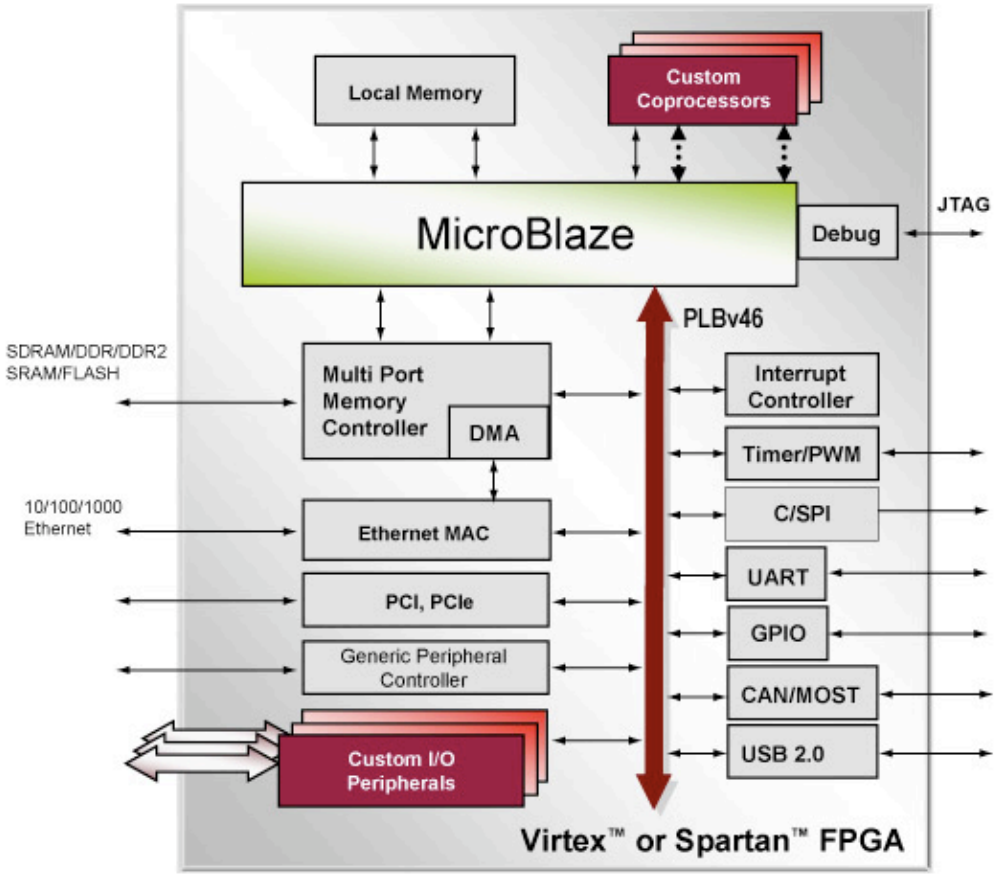
- MicroBlaze frente a otros Microprocesadores Embebidos



Microprocesador MicroBlaze de Xilinx

Introducción MicroBlaze

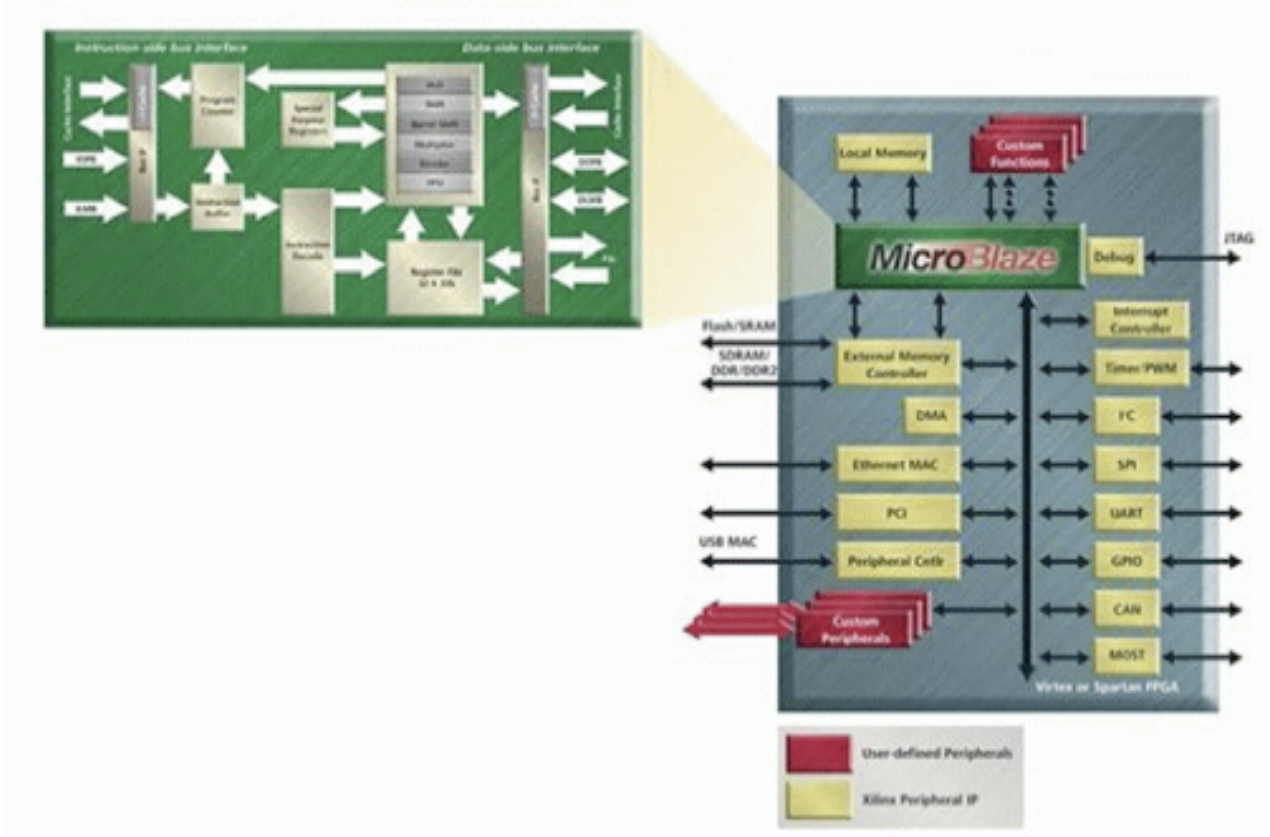
- Arquitectura de MicroBlaze



Microprocesador MicroBlaze de Xilinx

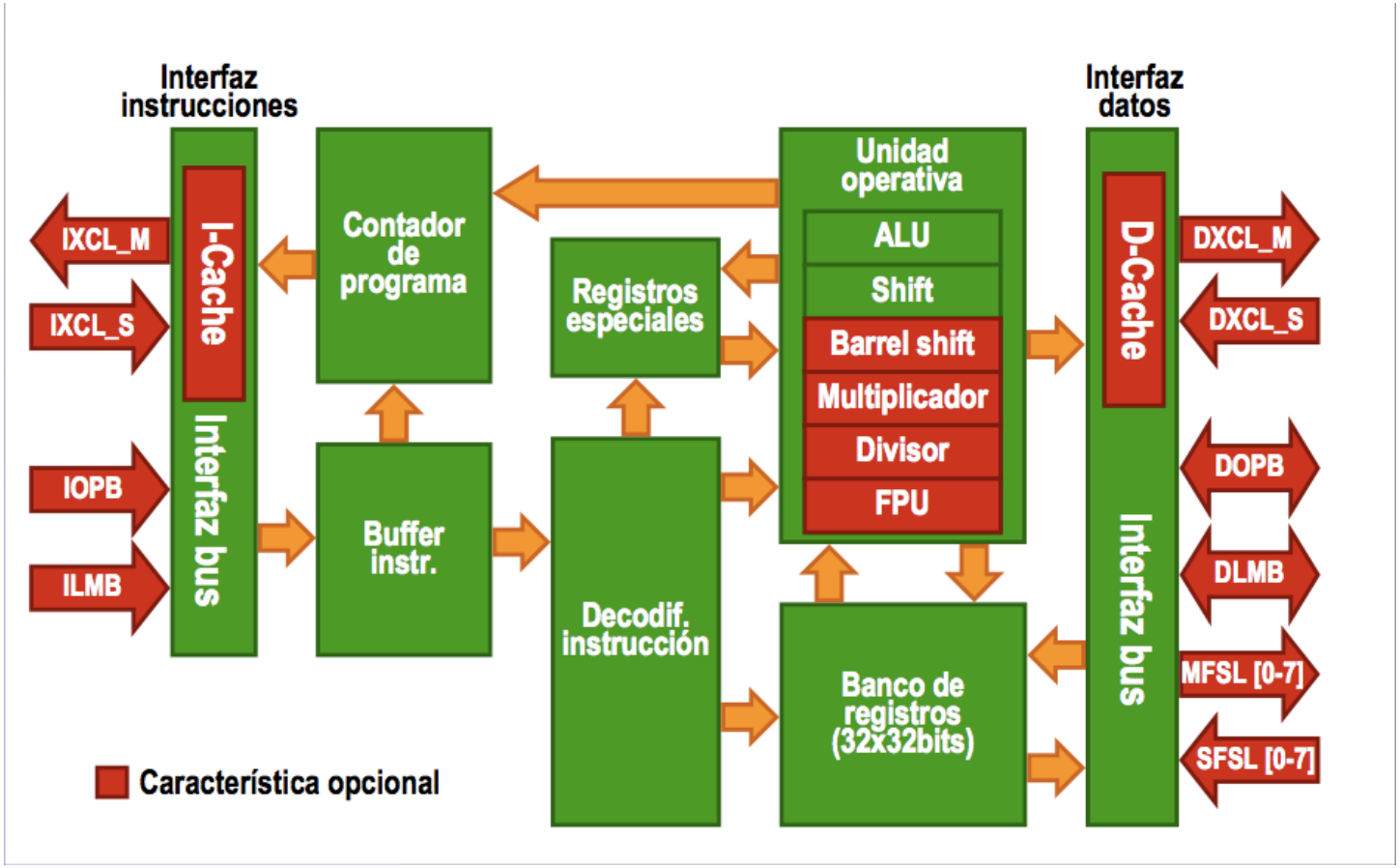
Introducción MicroBlaze

- Arquitectura de MicroBlaze



Introducción MicroBlaze

- Arquitectura de MicroBlaze



Introducción MicroBlaze

- Características fijas de MicroBlaze
  - Presenta un núcleo de 32 bits.
  - Buses de direcciones y datos de 32 bits.
  - Instrucciones de 3 operandos y 2 modos de direccionamiento (directo e inmediato).
  - 32 registros de propósito general (R0-R31)
    - Se ponen a cero cuando se configura la FPGA.
    - No cambian con las señales de Reset.
  - Pipeline de 5 etapas.
    - La 1ª instrucción tarda en ejecutarse 5 ciclos de reloj para evitar errores de sincronización, las demás instrucciones se ejecutan en un ciclo de reloj.
  - Formato Big-Endian (b0…b31) (MSB…LSB)

Introducción MicroBlaze

- Características fijas de MicroBlaze
  - Unidad aritmético-lógica (ALU)
    - Multiplicación y división entera (hardware)
    - Barrel shifter (desplazamiento bits con sólo una instrucción, hardware)
  - Unidad de punto flotante (FPU)
    - Estándar IEEE 754
    - Suma, resta, multiplicación, división y comparación.
  - Interfaz de depuración hardware (MDM)
  - Excepciones hardware.
  - Interfaces FSL.
  - Caché de instrucciones y datos.

Microprocesador MicroBlaze de Xilinx

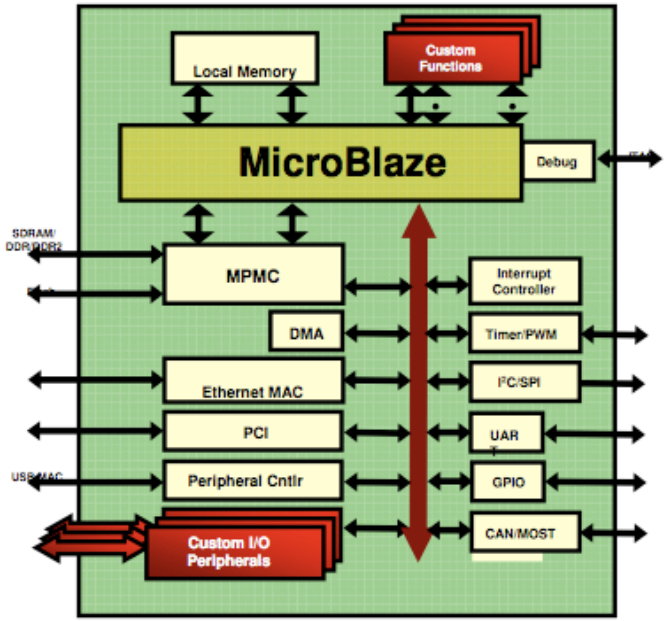
Introducción MicroBlaze

- Características generales de MicroBlaze

Processor	
Processor (CPU) Type	32-bit
Processor Architecture	MicroBlaze - RISC processor
MMU	Memory Management Unit
Processor Frequency	25 Mhz to 200 Mhz

Peripherals (IP cores)	
Memory	On-Chip RAM, Flash, SRAM, SDRAM, DDR
Basic IP Peripherals	Timer, Interrupt Controller, UART, GPIO
Advanced IP Peripherals	TEMAC, CAN, MOST, USB, PCIe, MultiPort Memory Controller, ***Custom***

Software	
RTOS	Nucleus, ThreadX, uClinux, uC/OS-II, uITRON
Languages	Assembly, C/C++
Tools	EDK & Platform Studio, GNU Tools, Eclipse IDE, Nucleus EDGE, Lauterbach T32, Computex F-Sight, LynuxWorks Luminosity, Agilent FPGA DynamicProbe



Virtex or Spartan FPGA

Microprocesador MicroBlaze de Xilinx

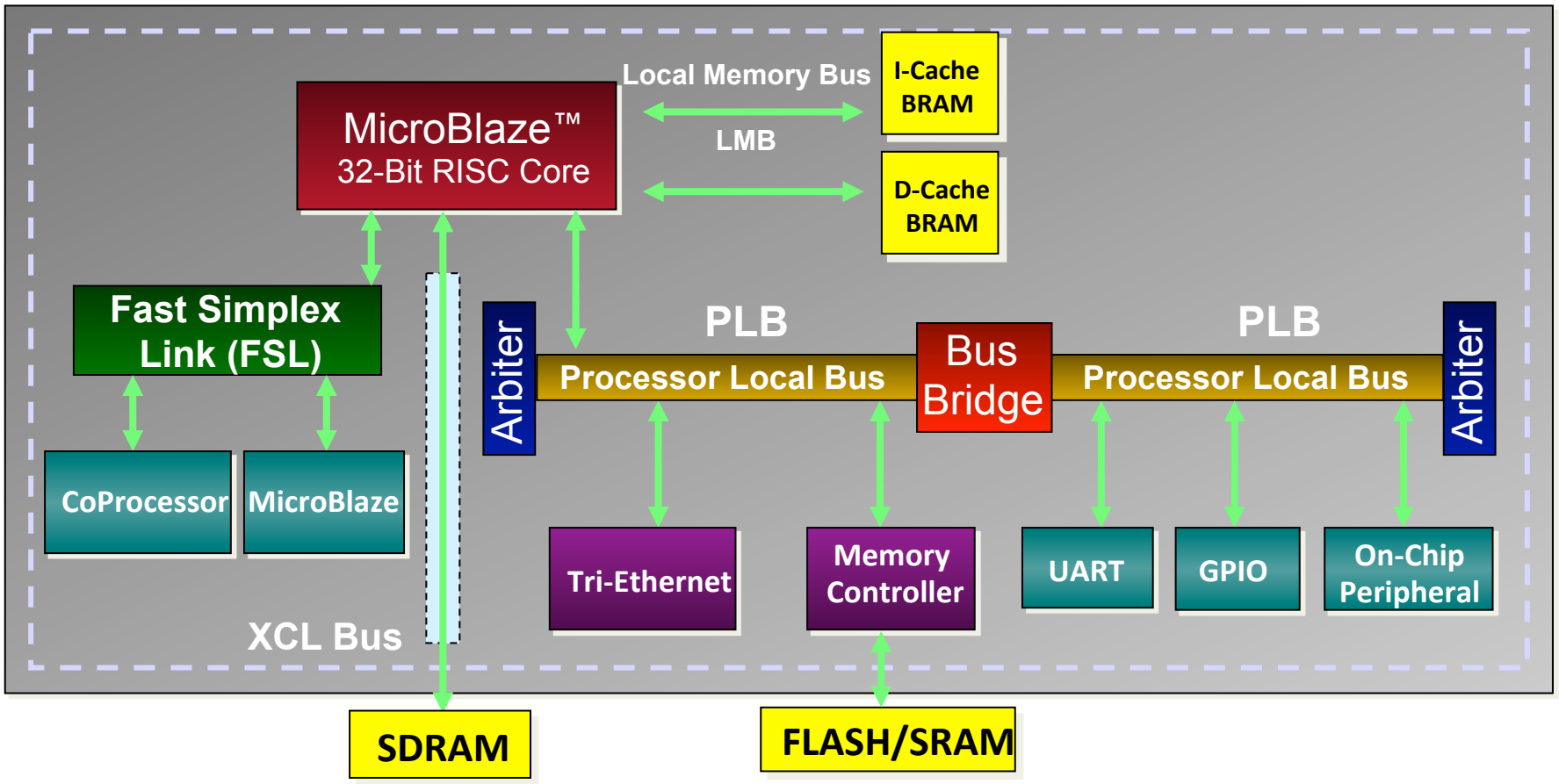
Introducción MicroBlaze

- Evolución de MicroBlaze

Features	MicroBlaze versions						
	v1	v2	v3	v4	v5	v6	v7
Pipeline Depth	3	3	3	3	5	3 & 5	3 & 5
Max Integer Perf.	82 DMIPS	125 DMIPS	125 DMIPS	166 DMIPS	240 DMIPS	240 DMIPS	240 DMIPS
Local Memory	0 or 8 - 64 KB	0 or 8 - 64 KB	0 or 8 - 64 KB	0 or 2 - 128 KB	0 or 2 - 256 KB	0 or 2 - 256 KB	0 or 2 - 256 KB
Multiplier, Barrel Shifter	option	option	option	option	option	option	option
Divider	--	option	option	option	option	option	option
Coprocessor Interface	--	FSL	FSL	FSL	FSL	FSL	FSL
Instr. & Data Cache	--	0 or 8 - 64 KB	0 or 8 – 64 KB	0 or 2 – 64 KB	0 or 2- 64 KB	0 or 2- 64 KB 64B – 1024B uCache	0 or 2- 64 KB 64B – 1024B uCache
Cache Interface	--	--	Cache Link	Cache Link	Cache Link	Cache Link	Cache Link
Floating Point Unit	--	--	--	single precision 33 MFLOPS	single precision 50 MFLOPS	single precision 50 MFLOPS	single precision 50 MFLOPS
MMU	--	--	--	--	--	--	Option (MPU or MMU) Full Linux Support
Debug Interface	ROM monitor	JTAG HW Debug	JTAG HW Debug	Debug + Trace	Debug + Trace	Debug + Trace	Debug + Trace
Primary FPGA Targets	Virtex II Spartan 2	Virtex II Pro Spartan 2E	Virtex II Pro Spartan 3	Virtex 4 Spartan 3E	Virtex 5	Virtex 5 family Spartan 3 family	Virtex 5 family Spartan 3 family

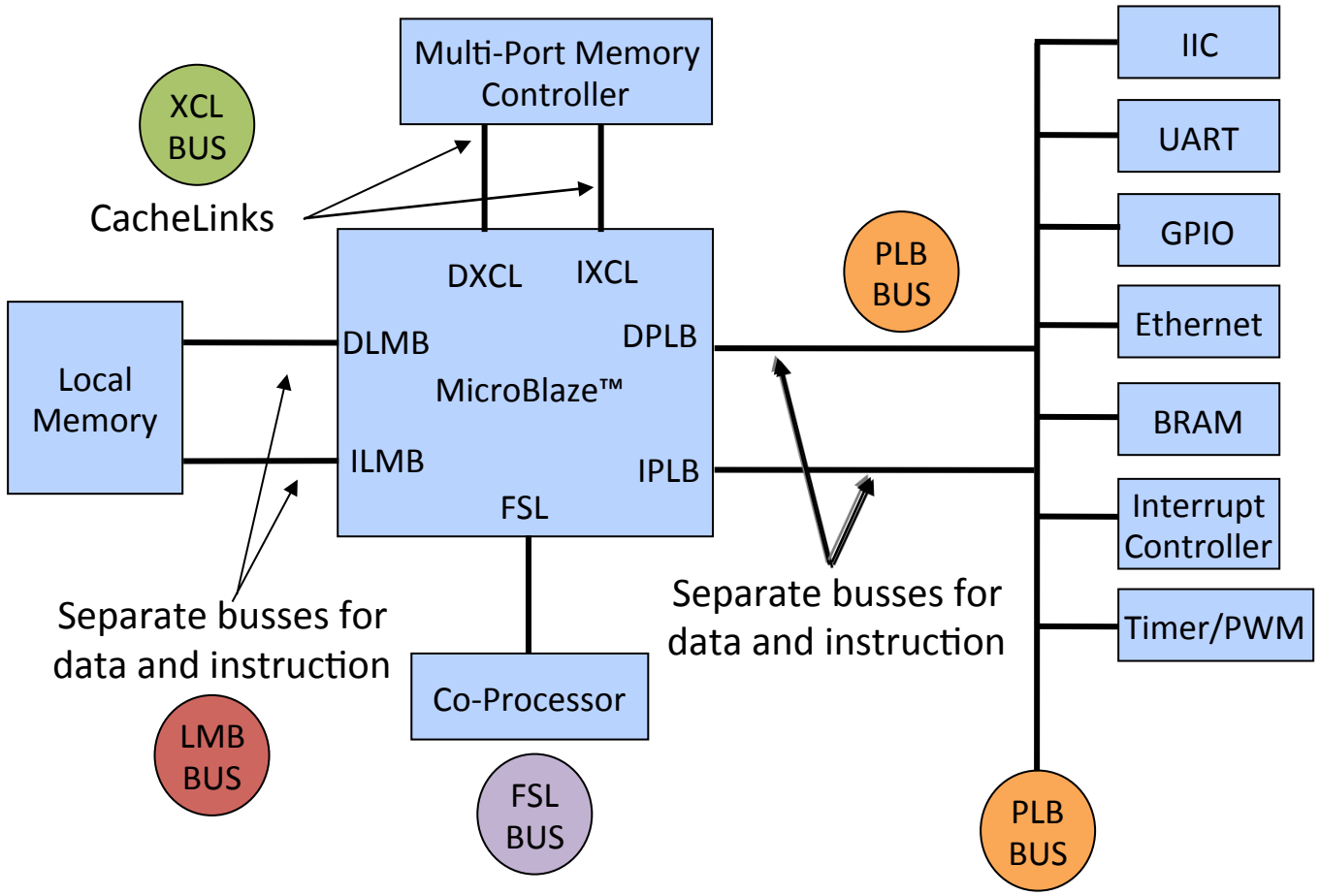
Buses e interfaces

- Esquema de buses en MicroBlaze

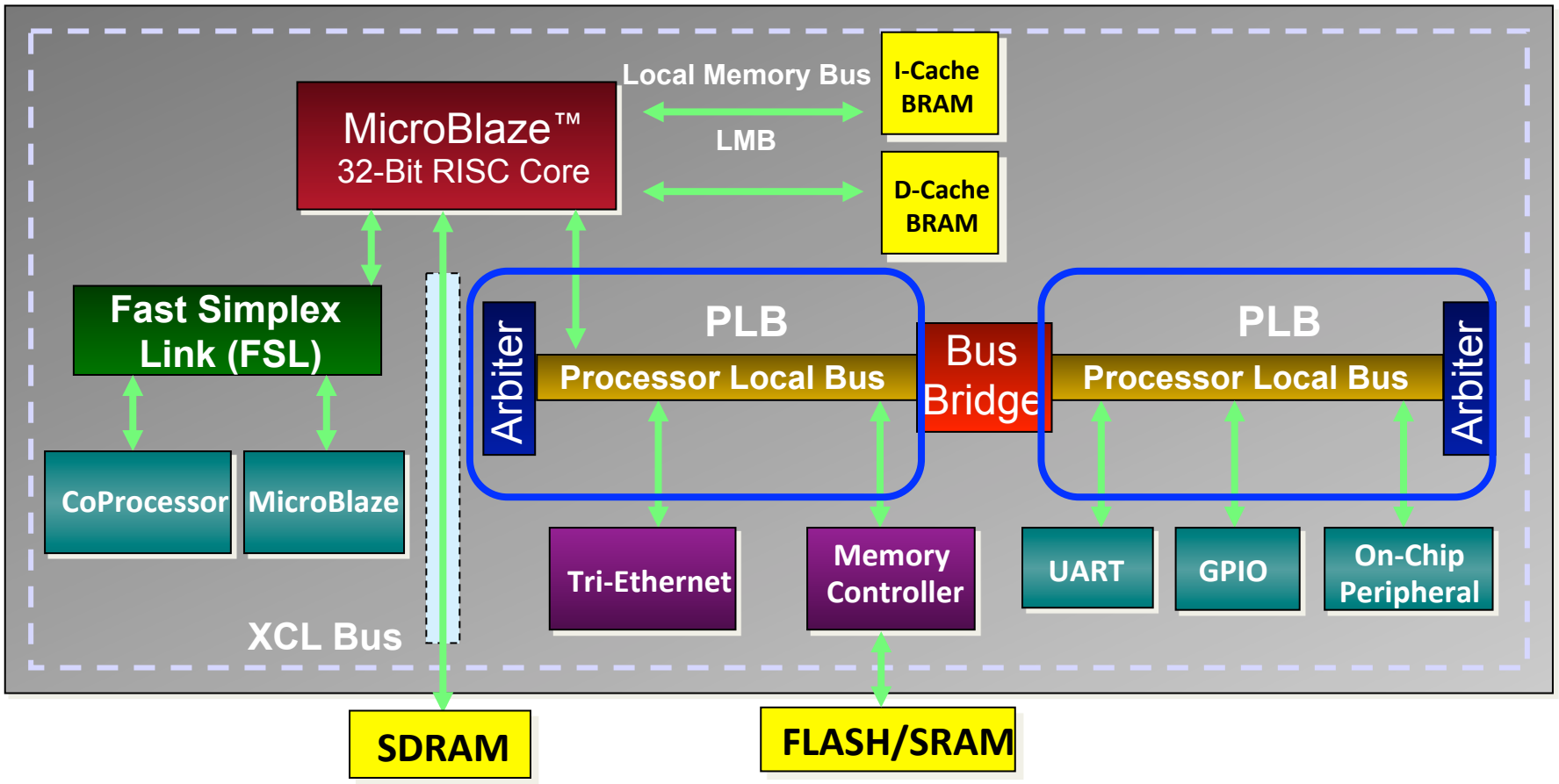


Buses e interfaces

- Esquema de buses en MicroBlaze



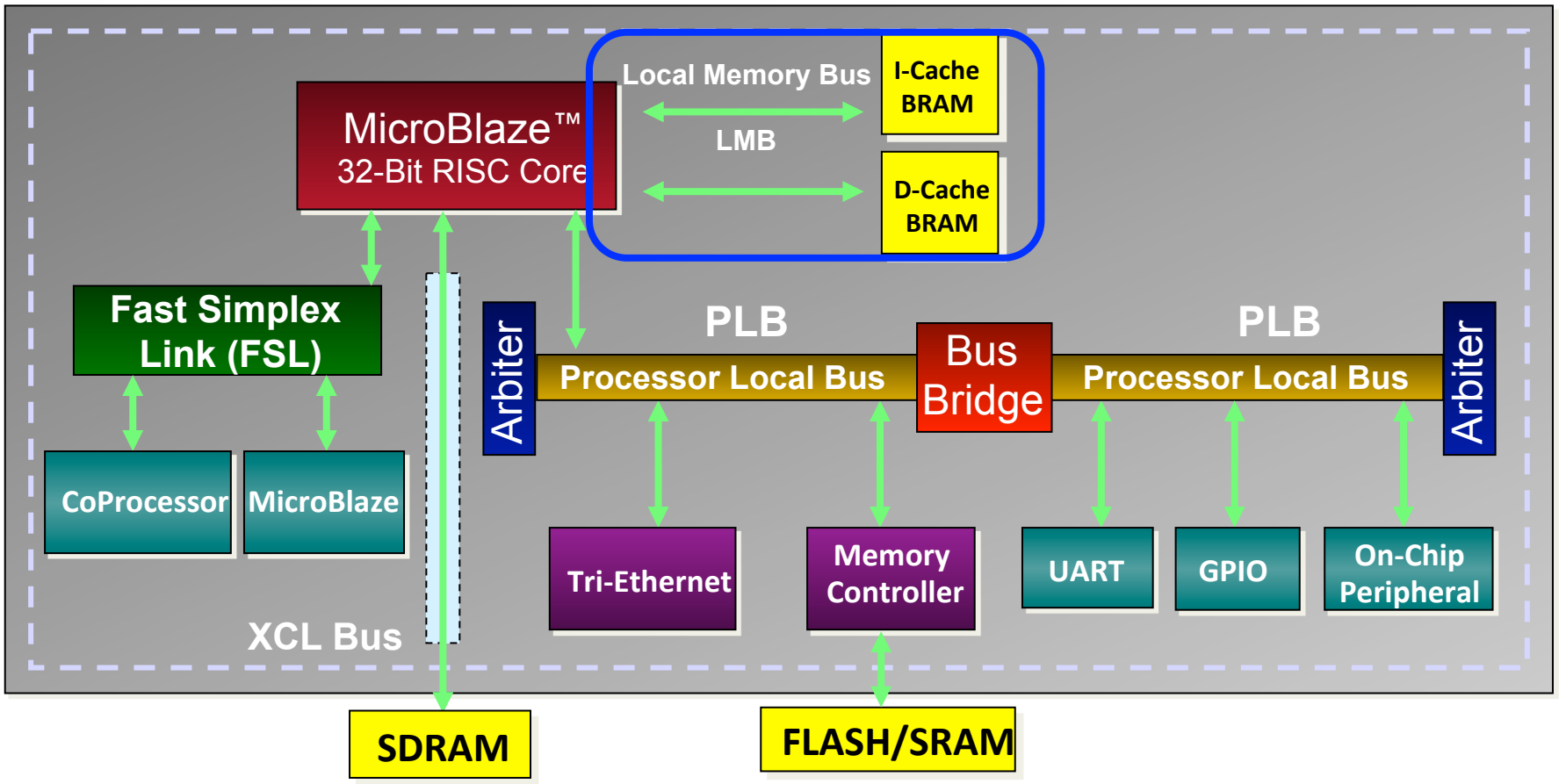
- Processor Local Bus (PLB)



Buses e interfaces

- Processor Local Bus (PLB)
  - Estándar de arquitectura de bus CoreConnect de IBM (instrucciones y datos)
  - Está formado por:
    - Maestros (16 máximo)
    - Esclavos (sin límite aunque Xilinx recomienda 16)
    - Árbitro de bus.
    - Interconexiones de bus.
  - El árbitro de bus recibe las peticiones de bus de los maestros y cede el bus a uno de ellos.
    - Prioridades fija y dinámica.
  - Las interconexiones de bus se realizan mediante bridges.

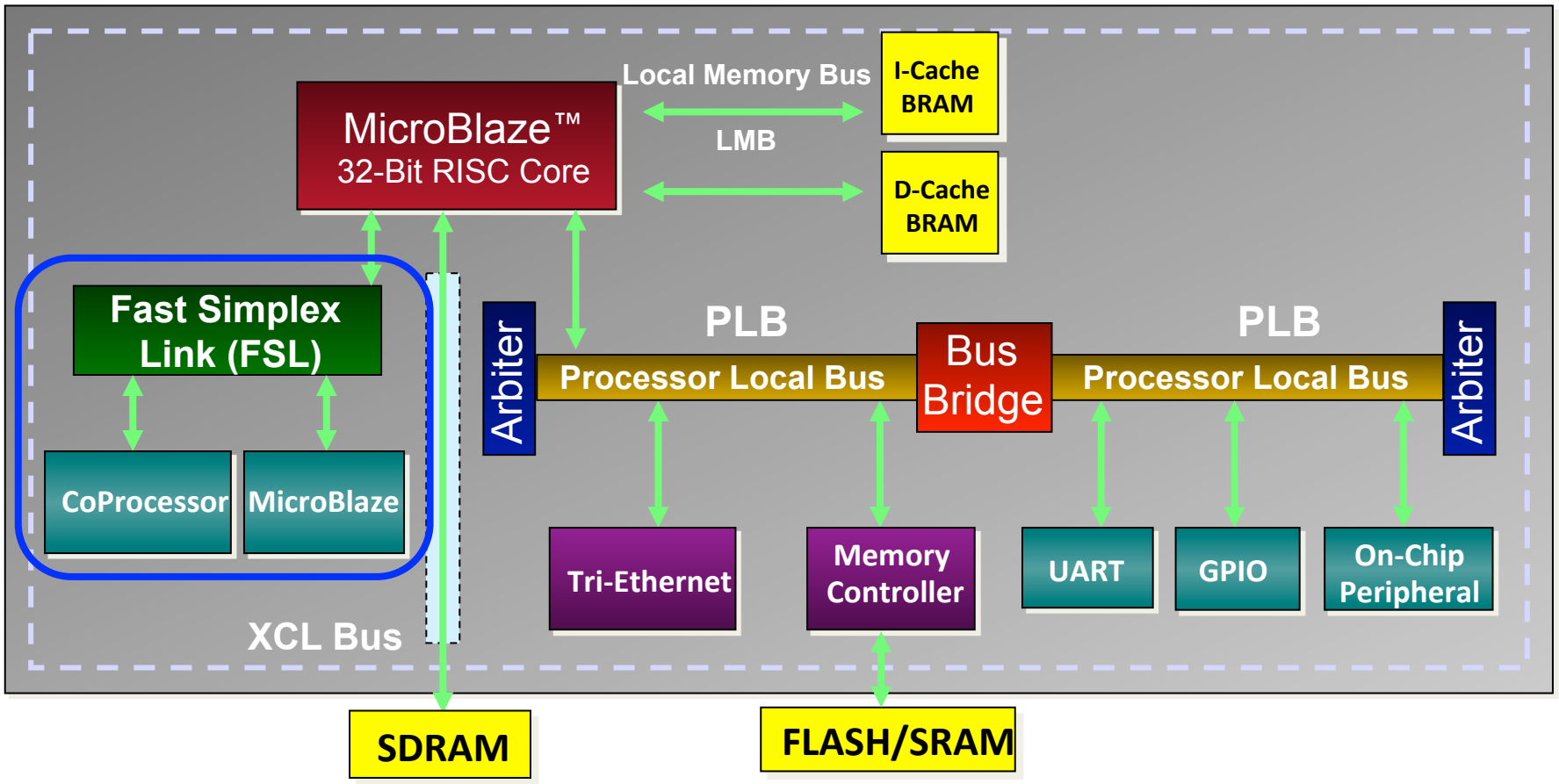
- Local Memory Bus (LMB)



Buses e interfaces

- Local Memory Bus (LMB)
  - Acceso en un solo ciclo de reloj a memoria dedicada (BRAM) dentro de la FPGA
    - ILMB: Interfaz de instrucciones.
    - DLMB: Interfaz de datos.
  - Mínimo número de señales de control y protocolo síncrono simple para transferencias de información eficientes.
  - Más rápido que el Bus PLB (125 MHz) en todos los dispositivos.

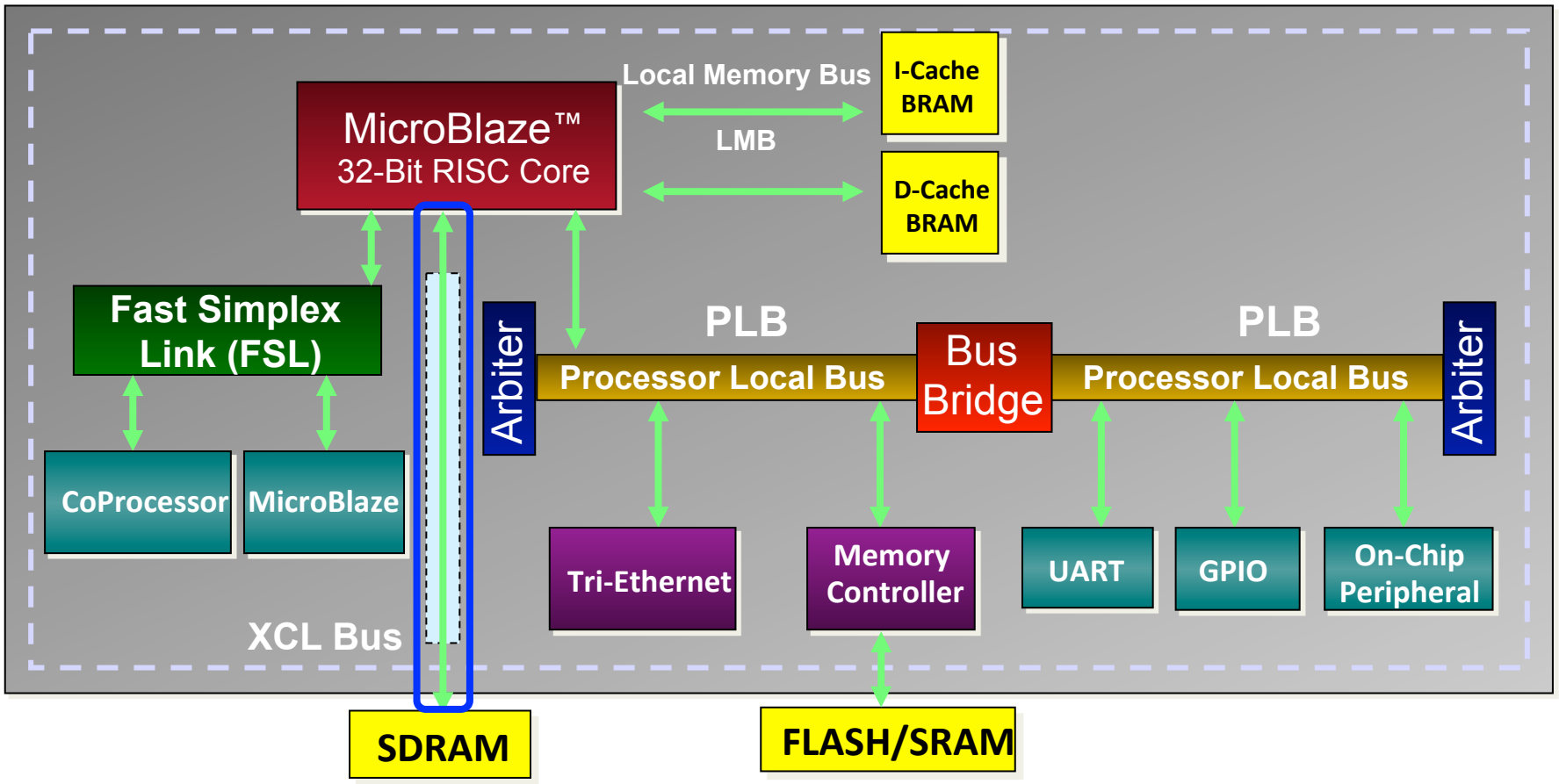
- Fast Simplex Links (FSL)



Buses e interfaces

- Fast Simplex Links (FSL)
  - Interfaces unidireccionales punto a punto dedicados y sin arbitraje basados en memoria FIFO.
  - Acceso rápido, 2 ciclos de reloj en Microblaze.
  - Permiten acceso directo a los registros de propósito general desde coprocesadores hardware utilizando instrucciones en C y ensamblador.
  - Un máximo de 8 entradas y 8 salidas.
  - Tamaño de FIFO configurable.
  - Reloj de la FIFO síncrono o asíncrono.

- Xilinx Cache Link (XCL)

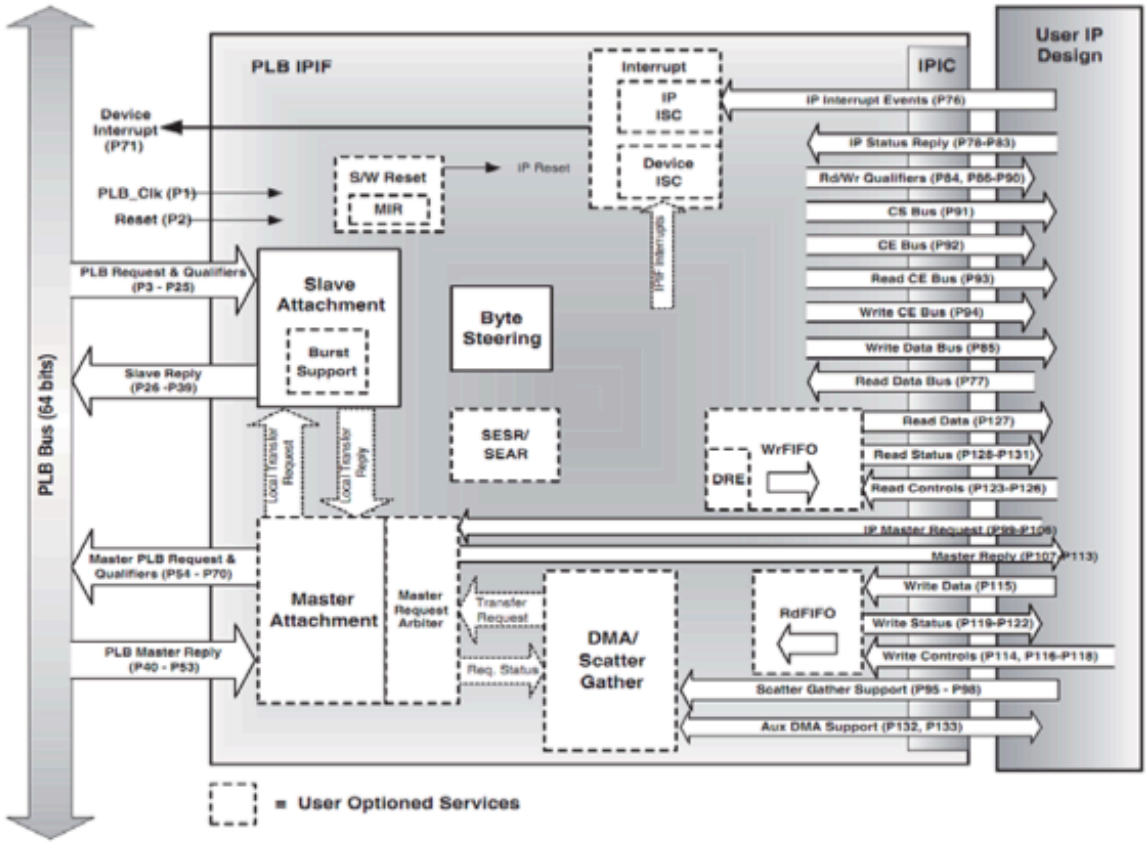


Buses e interfaces

- Xilinx Cache Link (XCL)
  - Diseñado para conectar directamente memorias externas a MicroBlaze mediante buffers tipo FSL.
  - Necesario habilitar el uso de caché en MicroBlaze para su uso.
  - Control de hasta 8 memorias externas tipo DDR.
  - Usa un controlador de memorias diseñado por Xilinx.

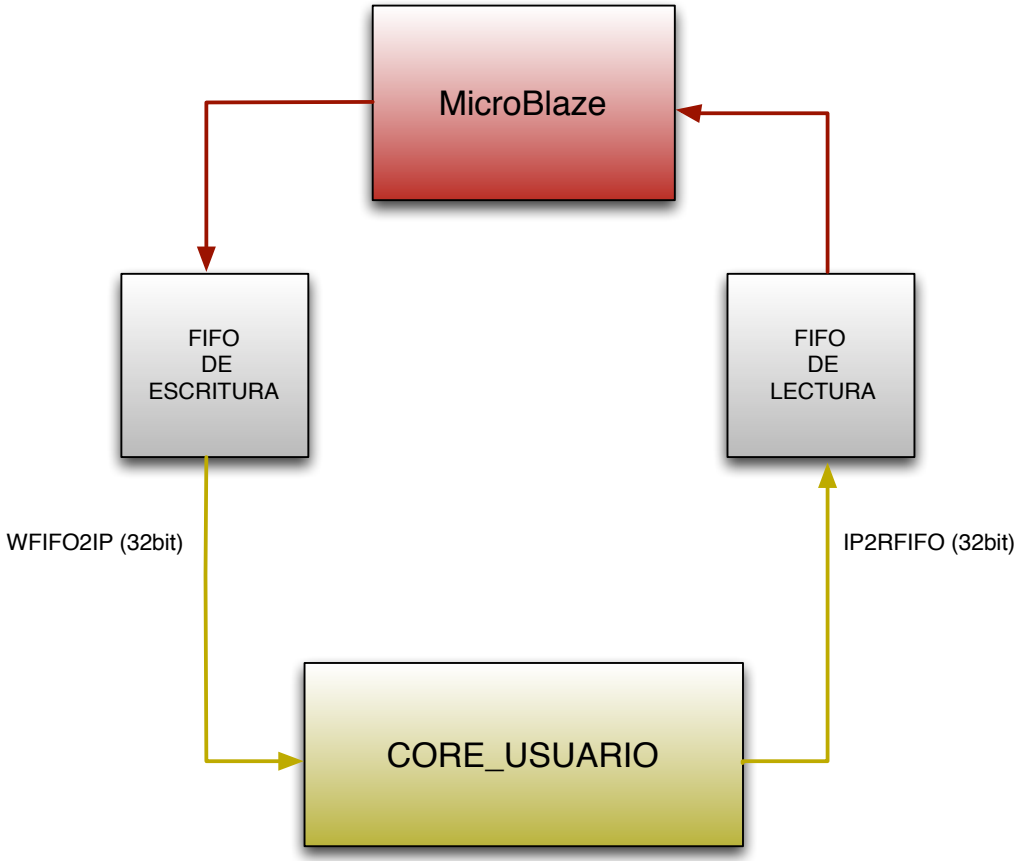
Periféricos de usuario

- Creación e importación de un periférico de usuario



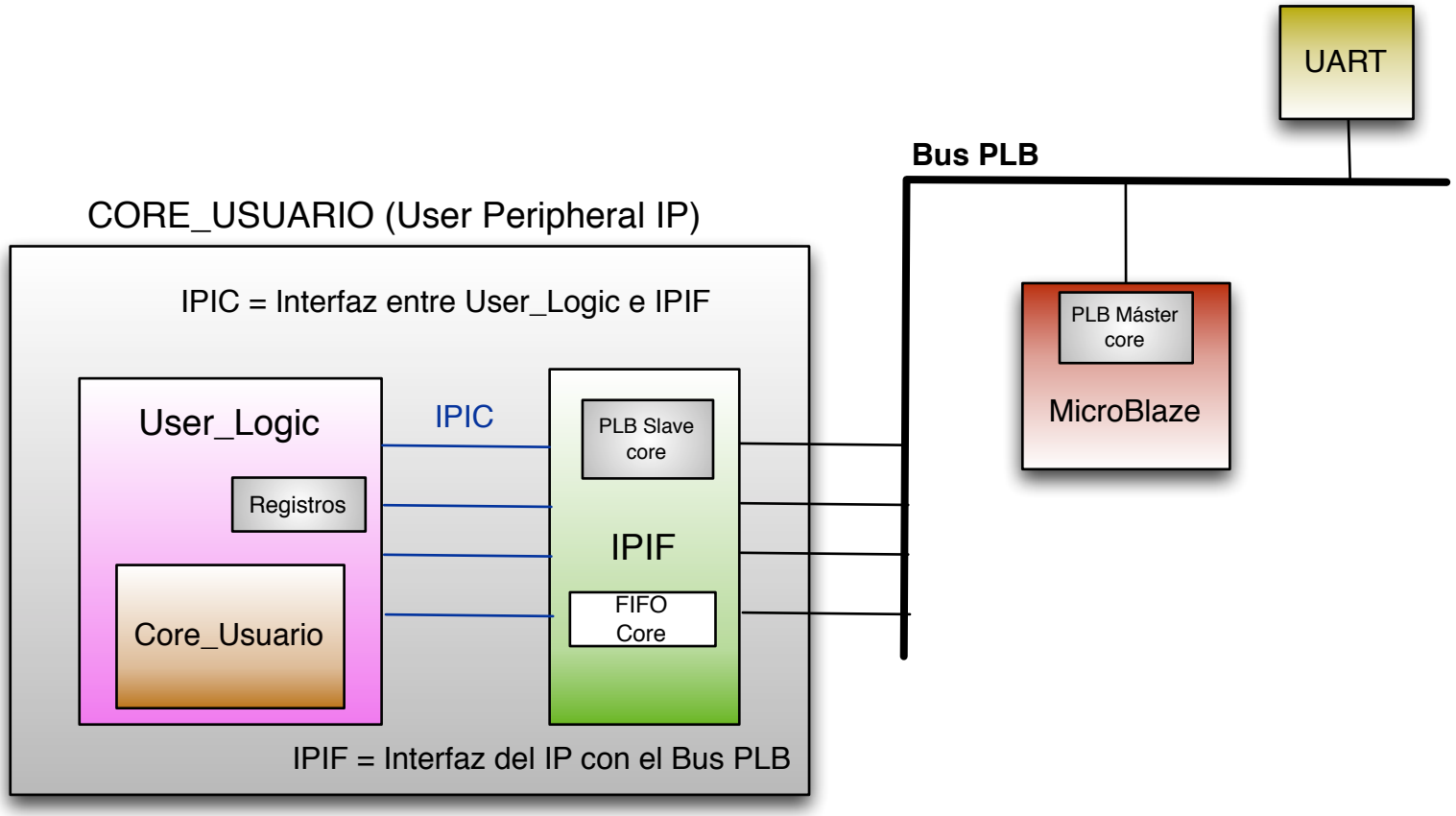
Periféricos de usuario

- Creación e importación de un periférico de usuario



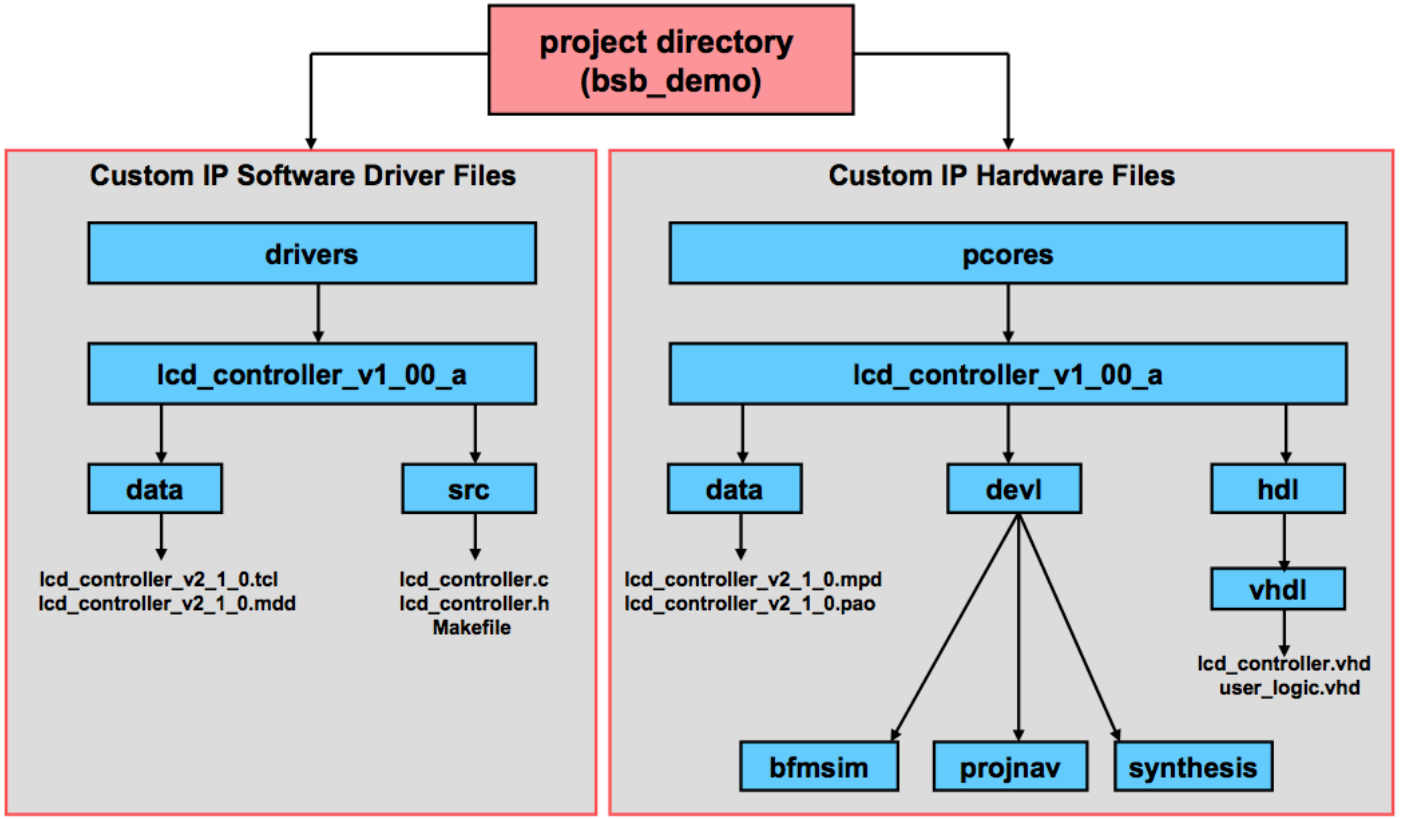
Periféricos de usuario

- Creación e importación de un periférico de usuario



Periféricos de usuario

- Creación e importación de un periférico de usuario



Periféricos de usuario

- Se generan varios archivos al crear la estructura del periférico:
  - Microprocessor Peripheral Definition (MPD)
    - Define la interfaz del periférico. Tiene la lista de puertos, la lista de conexiones con el bus y los parámetros básicos.
  - Peripheral Analysis Order (PAO)
    - Contiene la lista de archivos HDL que son necesarios para sintetizar el periférico creado, también define el orden en que son compilados y simulados.
  - User_Logic (.vhd)
    - Plantilla en VHDL para que el usuario pueda editar y añadir código. Se suele usar para implementar componentes de usuario que completan el diseño del periférico.
  - Core_Usuario (.vhd, .c, .h)
    - Son los archivos principales del periférico creado.
    - En el .vhd se pueden definir los puertos externos del mismo.
    - El .c tiene una estructura para introducir código y el .h las funciones a usar.

Ejercicio

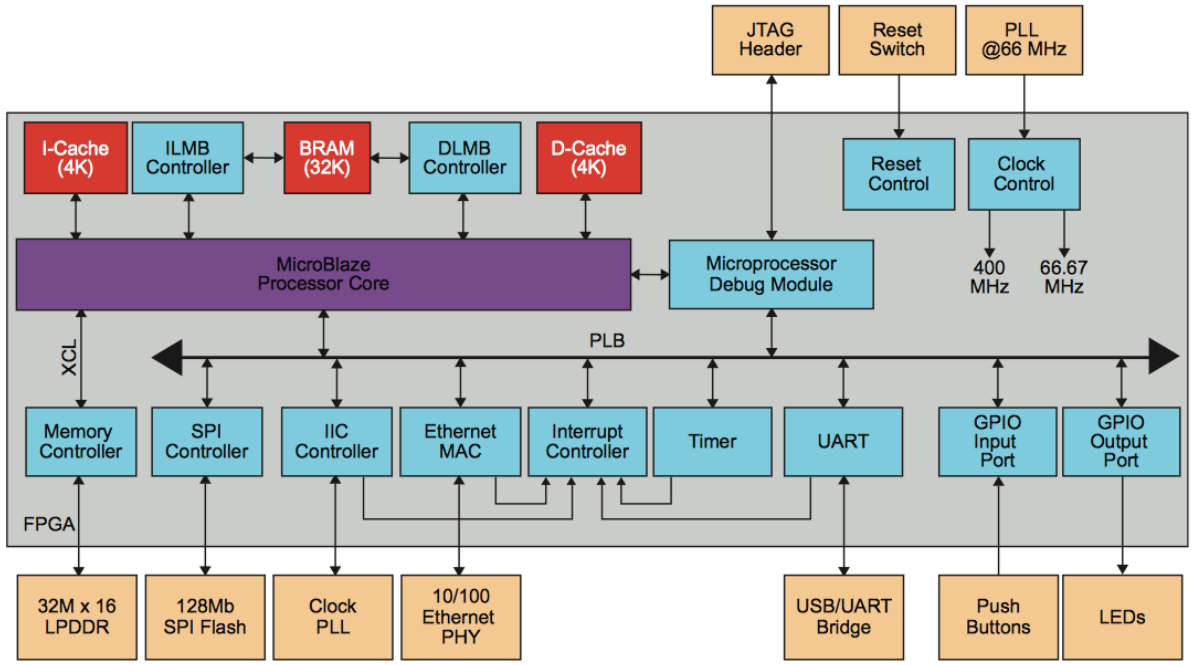
6

Temporizador e Interrupciones

En este sexto diseño con MicroBlaze vamos a aprender a utilizar un timer para implementar un contador y, adicionalmente, a aprender como funciona el sistema de interrupciones en MicroBlaze.

El desarrollo de este ejercicio nos llevará a crear un diseño hardware con Microblaze utilizando de nuevo BSB con los periféricos habitulaes pero añadiendo también un periférico de tipo XPS Timer que nos establecerá una base de tiempos.

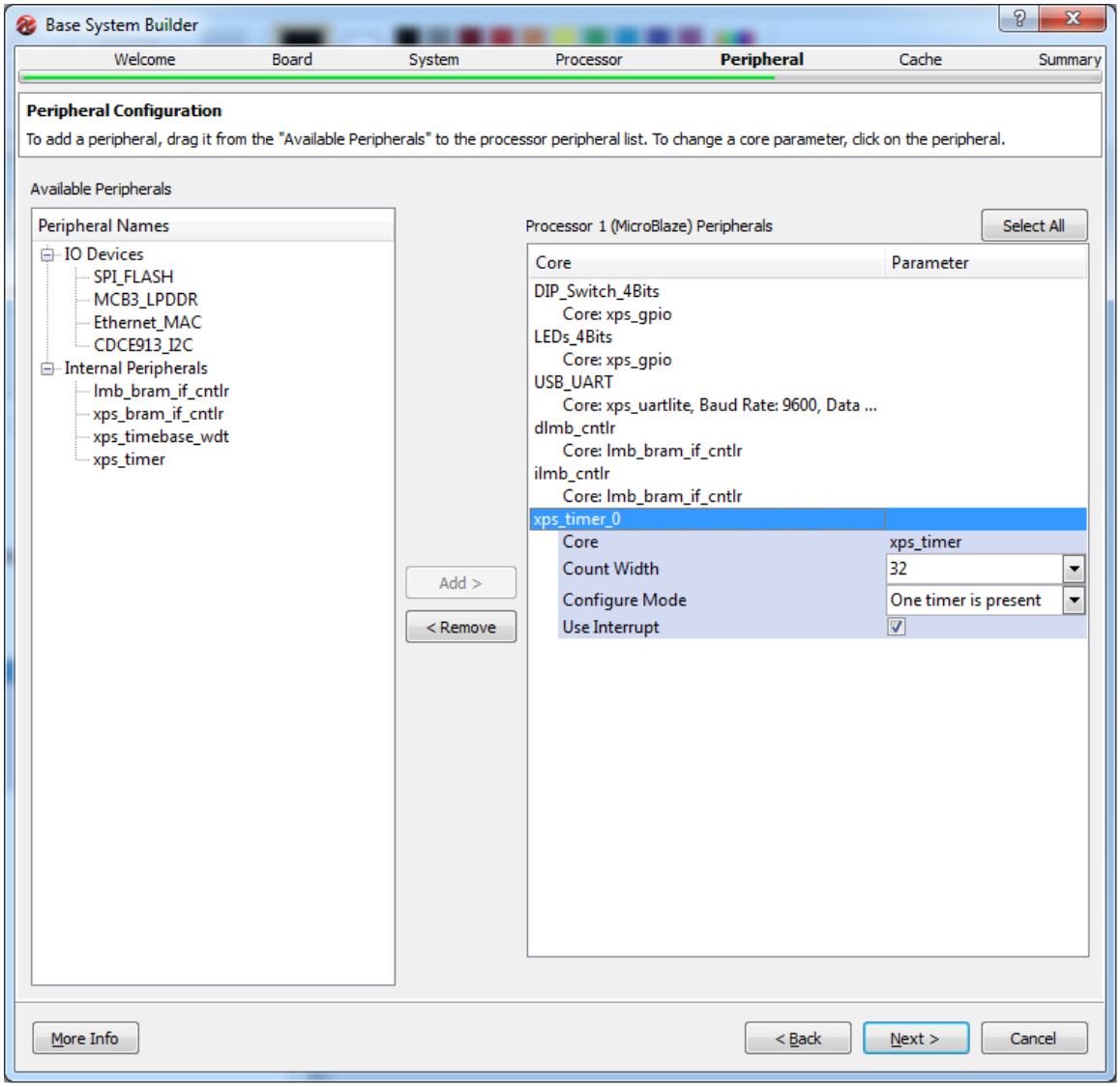
Así mismo, crearemos una aplicación software para gestionar este timer, lo que nos permitirá aprender como funciona el sistema de interrupciones en MicroBlaze. El diagrama de bloques que tendrá esta aplicación es el siguiente.



Creación sistema empotrado - XPS

Siguiendo los pasos del Ejercicio_1, creamos un nuevo proyecto con el nombre Ejercicio_6. Definiendo MicroBlaze como nuestro microprocesador empotrado, la misma placa de desarrollo usada anteriormente, la misma frecuencia y el mismo tamaño de memoria BRAM.

En este caso, usaremos el puerto USB_UART para poder visualizar el flujo de comunicación. También emplearemos los LEDs y los switches, y por último añadiremos un periférico XPS Timer, y lo configuraremos de acuerdo a la siguiente figura:

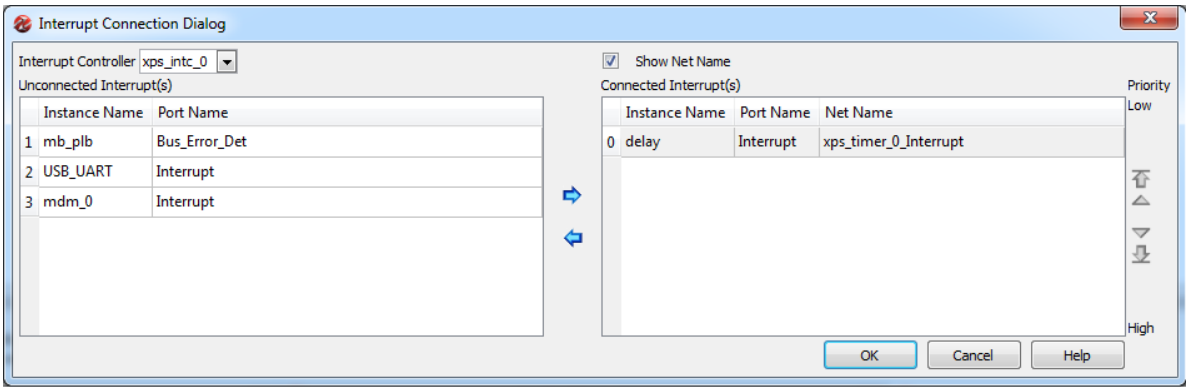


En la System_Assembly_View podemos observar como, aparte de nuestro periférico Timer, el asistente ha creado otro periférico llamado xps_intc, que es el que va a recoger todas las solicitudes de interrupción generadas por el sistema y gestionarlas de acuerdo a una lista de prioridades para que MicroBlaze las procese de manera adecuada.

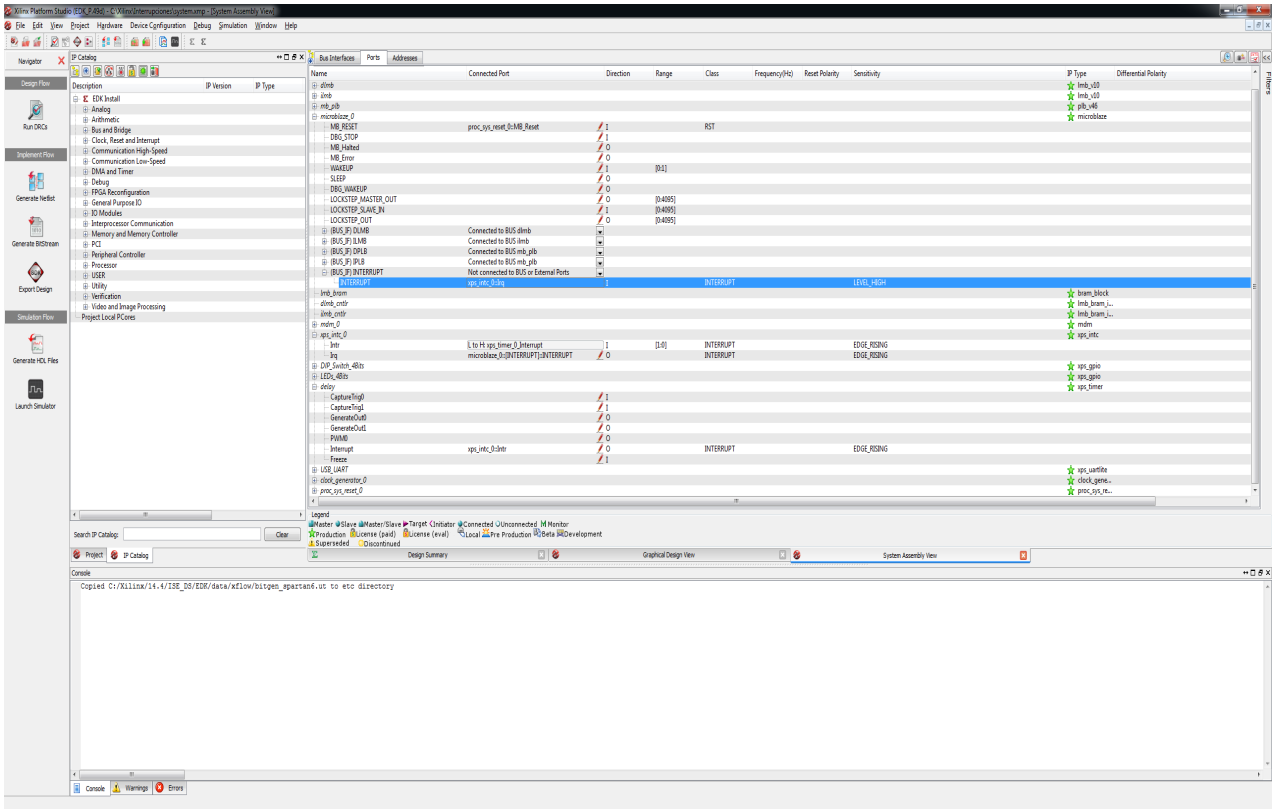
Cambiamos el nombre del componente xps_timer_0 por delay. Además, conectamos las siguientes señales del temporizador delay en la pestaña ports:

- CaptureTrig0 a net_gnd (basta seleccionarlo, indica que no tendremos arranque de timer por señal externa)
- Interrupt a xps_intc_0::Intr (ya debería estar seleccionado. Indica que la interrupción de este periférico debe dirigirse al controlador de interrupciones)

En el periférico xps_intc_0 vemos que la señal de interrupciones consiste en un desplegable en el que nos deja ordenar por prioridades todas las señales de interrupción que recibe este controlador. En este momento conectamos la única que tenemos, con lo que nos quedaría algo así:



La salida de este periférico ya debe estar conectada al puerto de interrupciones de MicroBlaze, por lo que finalmente el sistema nos quedará tal y como se muestra en la siguiente figura:

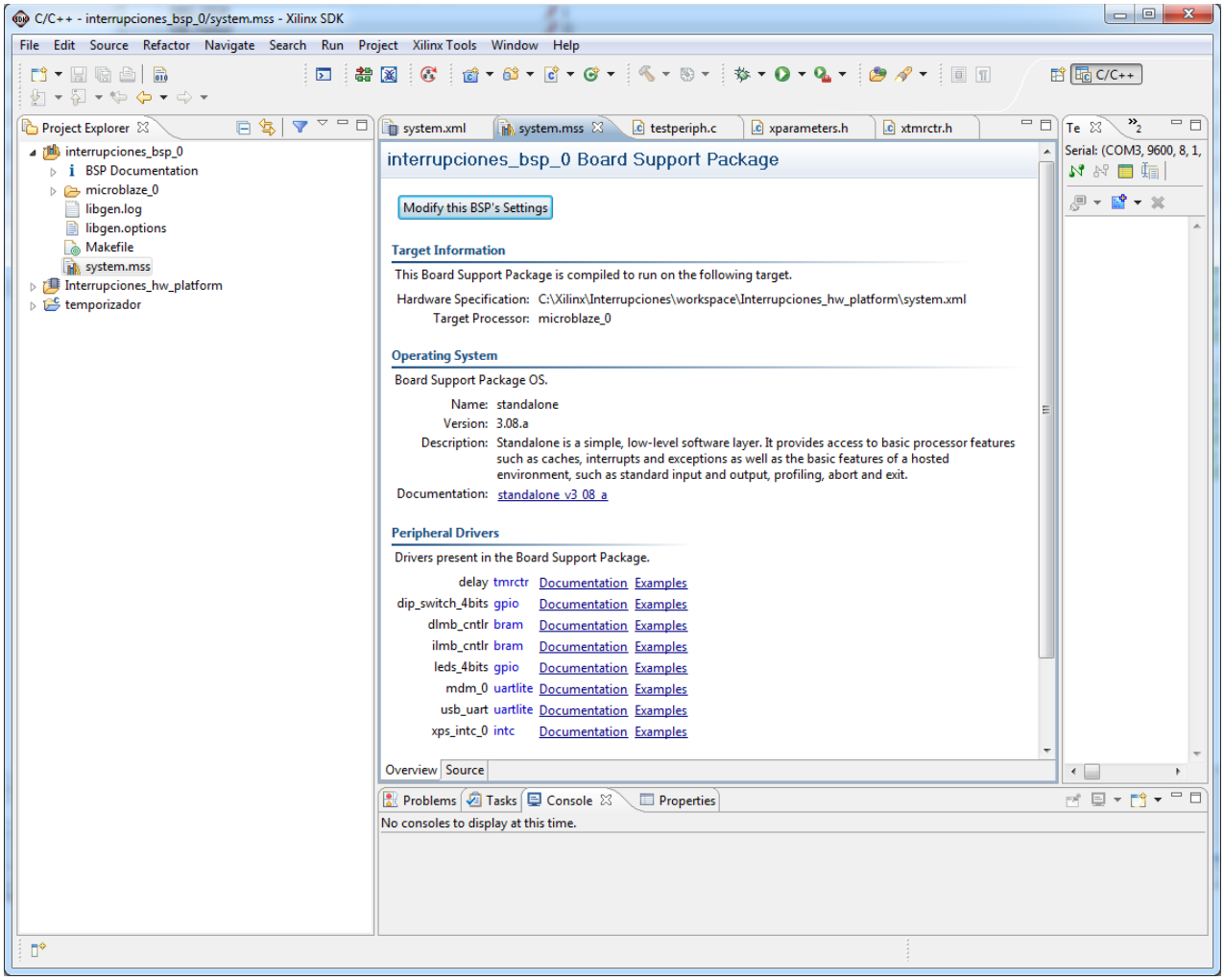


Una vez realizados todos estos cambios, procedemos a generar el bitstream, a exportarlo a SDK, generamos el correspondiente BSP y procedemos a implementar el test de periféricos para comprobar que todo funciona adecuadamente.

Antes de implementar nuestro código, debemos de entender el funcionamiento del periférico XPS_Timer. Para ello, busca la documentación proporcionada por Xilinx y localiza en ella la descripción de los registros, prestando especial atención al registro de control/status (TCSR0). Compara los parámetros que aparecen en ese registro con lo que ya conoces sobre el funcionamiento de un Timer.

http://www.xilinx.com/support/documentation/ip_documentation/xps_timer.pdf

Estudia en el system.mss del SDK los drivers creados por Xilinx para la utilización del timer. Estudia con atención las funciones, los parámetros, y los ficheros de ejemplo.



Ahora vamos a crear una nueva aplicación software, cuyo código nos viene dado en el fichero temporizador.c

```
#include "xparameters.h"
#include "xbasic_types.h"
#include "xgpio.h"
#include "xtmrctr.h"
#include "xintc.h"
#include "stdio.h"

XTmrCtr MyTimer, *MyTimerPtr;
XGpio MyLeds, *MyLedsPtr;
int LedBit;

int count = 0;
int flag_5_sec = 0;

void timer_int_handler(void *TimerPtr)
{
    XGpio_DiscreteWrite(MyLedsPtr, 1, 1 << LedBit);
    LedBit++; if (LedBit == 4) LedBit = 0;
    count++;
    if (count%5 == 0) flag_5_sec = 1;

}

int main(void)
{
    XIntc MyIntc, *MyIntcPtr;

    // Configure interrupt controller
    MyIntcPtr = &MyIntc;
    if (XIntc_Initialize(MyIntcPtr, XPAR_INTC_0_DEVICE_ID) !=
XST_SUCCESS)
    return XST_FAILURE;
    XIntc_Connect(MyIntcPtr, 0, timer_int_handler, MyTimerPtr);
    XIntc_Start(MyIntcPtr, XIN_REAL_MODE);
    XIntc_Enable(MyIntcPtr, 0);

    // After configuring the controller, interrupts can be enables
    microblaze_enable_interrupts();

    // Initialize timer
    MyTimerPtr = &MyTimer;
    if (XTmrCtr_Initialize(MyTimerPtr, XPAR_DELAY_DEVICE_ID) !=
XST_SUCCESS)
    return XST_FAILURE;
    XTmrCtr_SetResetValue(MyTimerPtr, 0, 50000000);
    XTmrCtr_SetOptions(MyTimerPtr, 0,
        XTC_DOWN_COUNT_OPTION + XTC_AUTO_RELOAD_OPTION +
XTC_INT_MODE_OPTION);
```

```
    // Start timer
    XTmrCtr_Start(MyTimerPtr, 0);

    // Initialize GPIO
    MyLedsPtr = &MyLeds;
    if (XGpio_Initialize(MyLedsPtr, XPAR_LEDS_4BITS_DEVICE_ID) !=
XST_SUCCESS)
    return XST_FAILURE;
    XGpio_SetDataDirection(MyLedsPtr, 1, 0x0);
    XGpio_DiscreteWrite(MyLedsPtr, 1, 0x0);

    while(1) {
      if (flag_5_sec==1)
            flag_5_sec = 0;
            xil_printf("Han pasado 5 segundos\n\r");

      }
}
```

Este código crea una base de tiempos de 1 segundo, y va encendiendo los leds de manera secuencial cada segundo. Además, cada 5 segundos saca un mensaje por hyperterminal.

El código compila sin errores, pero su funcionamiento no es correcto dado que tiene dos errores, uno respecto al propio lenguaje C, y otro respecto a la forma de gestionar la interrupción. Además, la base de tiempos no es exactamente de 1 segundo. Localiza estos errores y corrígelos.

Como ejercicio adicional, se propone realizar un proyecto en el que el controlador de interrupciones controle tanto las interrupciones de un timer como de dos switches independientes. La cuenta en segundos aparecerá en el hyperterminal, y uno de los switches controlará el sentido de la cuenta mientras que el otro actuará como start/stop.

Ejercicio

7

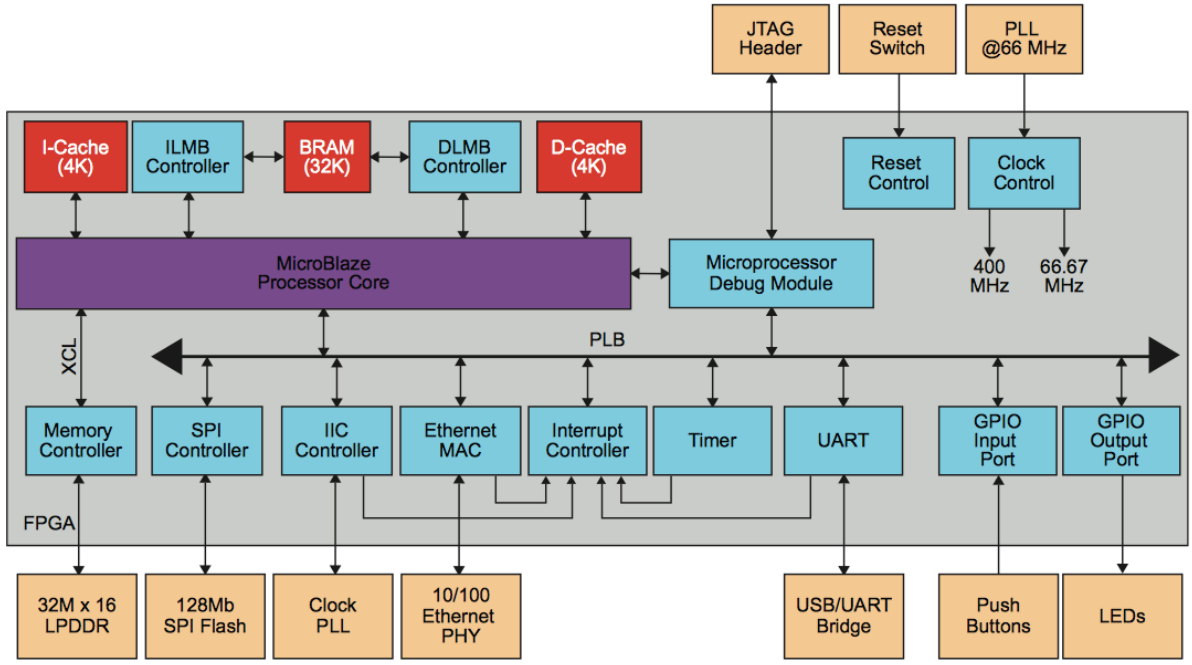
Integración de un Core como periférico

Si tenemos un Core diseñado previamente en ISE o con Simulink, podemos añadirlo a EDK según se va a explicar en este ejercicio.

A menudo, creamos Cores propios que hemos usado en diferentes proyectos y que podemos usar en cualquier nuevo desarrollo. El objetivo de este ejercicio es conocer como se puede integrar dicho Core dentro de un proyecto diseñado con MicroBlaze. Para dicha integración, usaremos el asistente para periféricos que incluye EDK.

El primer paso será crear un Core mediante la aplicación Xilinx Core Generator que incluye ISE y que nos genera un archivo .ngc

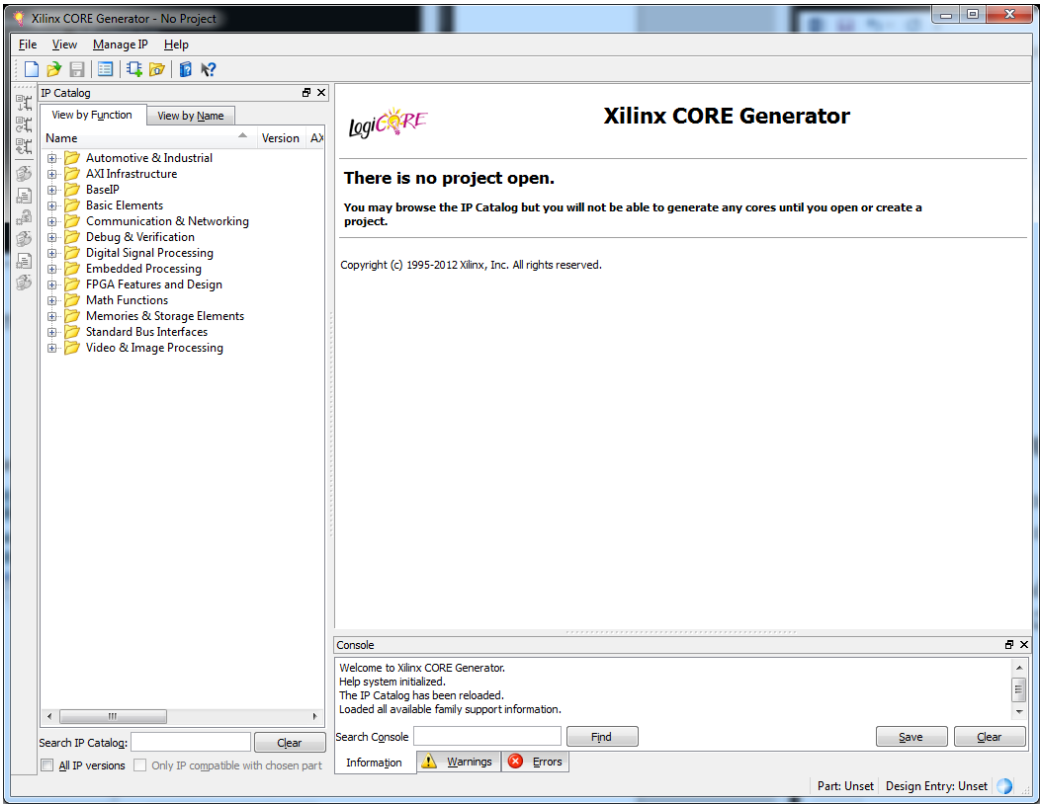
El diagrama de bloques que tendrá esta aplicación es el siguiente.



Creación de un Core

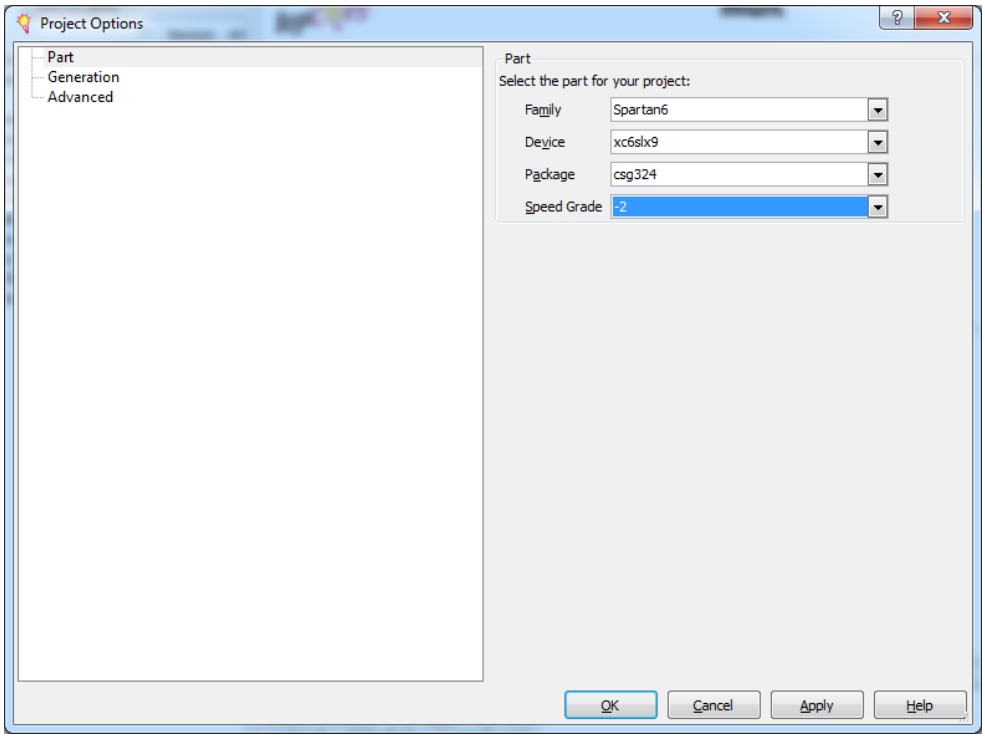
Como hemos comentado antes, vamos a usar la aplicación Xilinx Core Generator para crear un periférico que nos permita realizar la multiplicación de dos entradas de 16 bits y nos genere un resultado de 32 bits.

Desde el Menú de Xilinx, buscamos la aplicación comentada anteriormente. Se encuentra dentro de la carpeta de 32 o 64 bits (según corresponda) en ISE Design Tools.



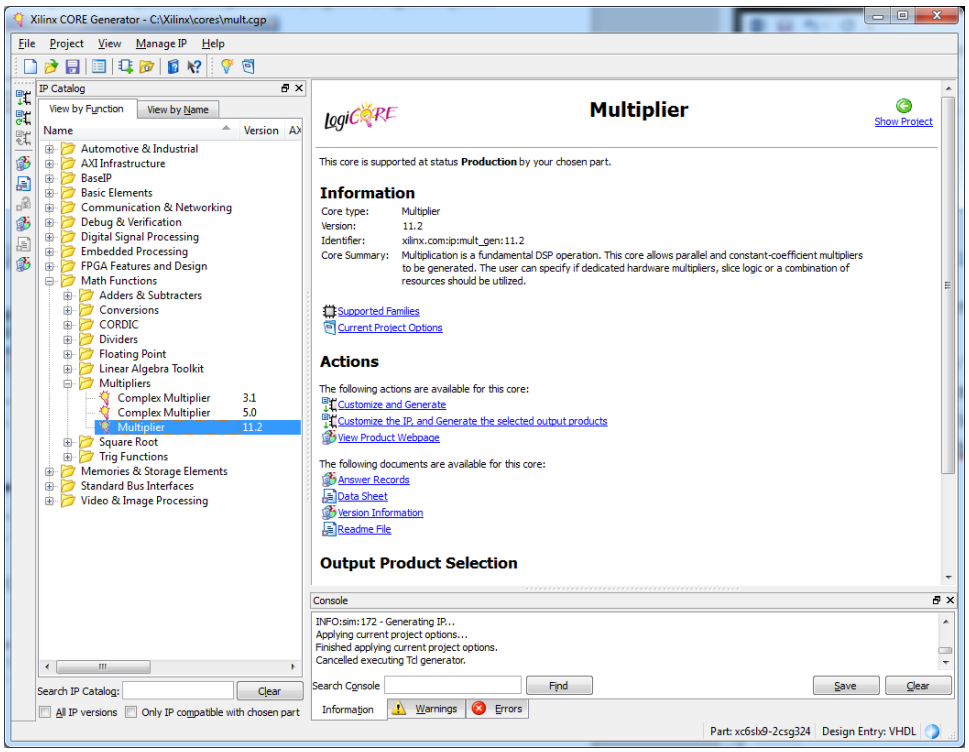
Seleccionando File->New Project escogemos donde lo queremos guardar. Lo habitual es crear una carpeta denominada Cores y almacenarlos todos ahí. Darle, por ejemplo, el nombre de mult.

Ahora debemos especificar con que FPGA vamos a utiliza el Core.

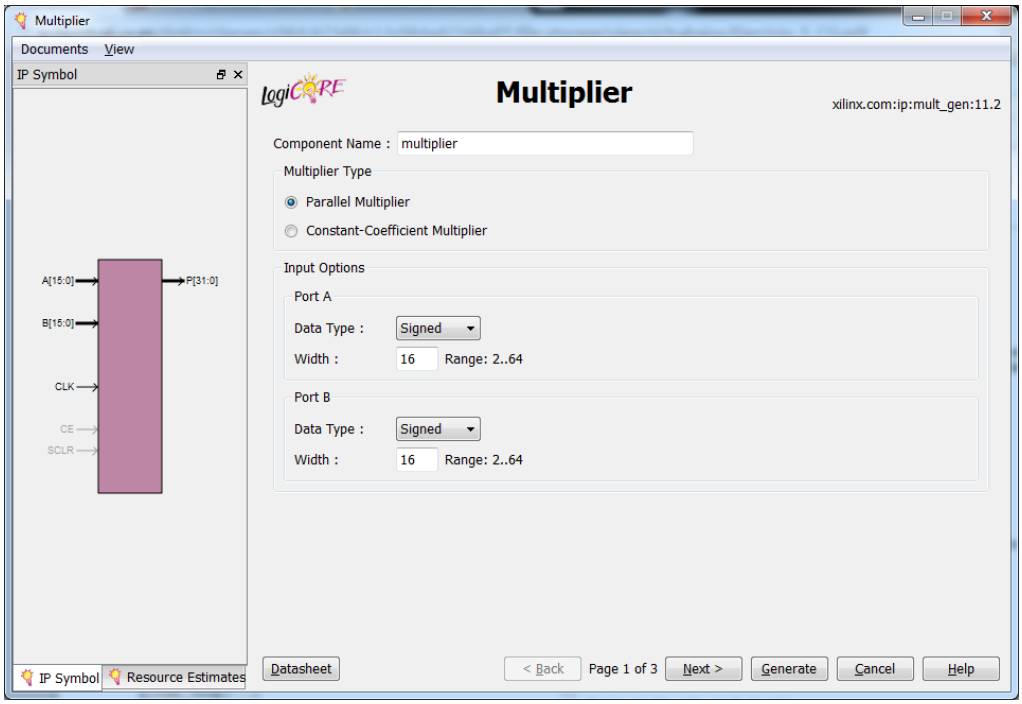


El resto de opciones las dejamos por defecto.

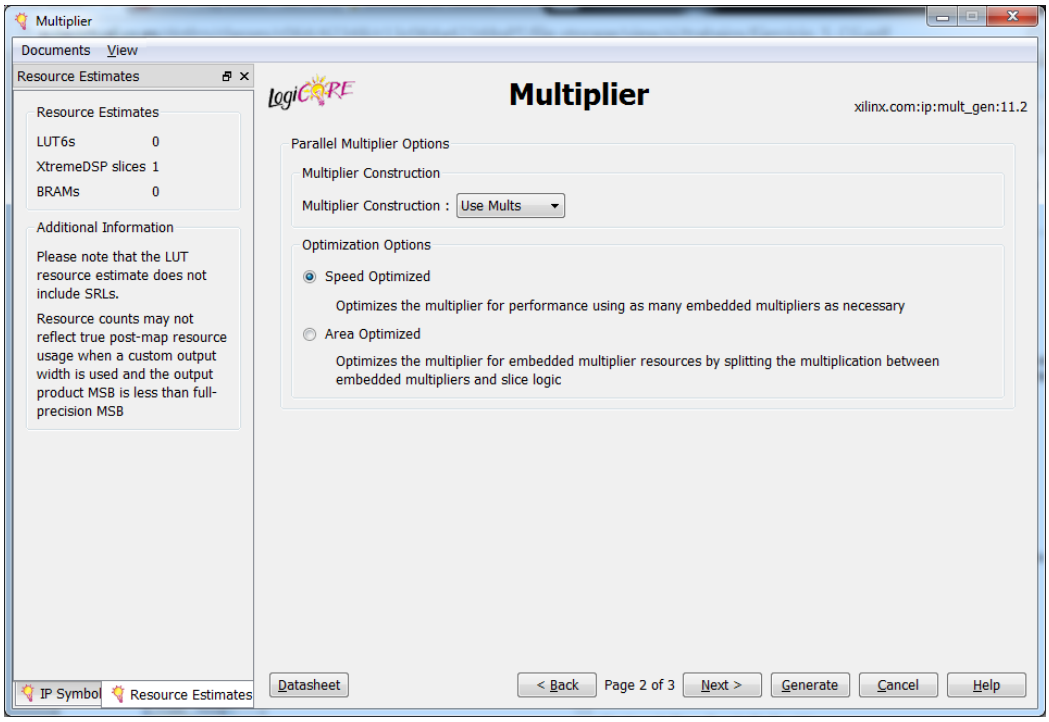
Una vez creado el Proyecto, debemos escoger que función es la que va a realizar. Antes de esto, explorar las diversas opciones de periféricos que nos presenta el CoreGenerator. Para implementar un core con la funcionalidad de multiplicador, entramos en Math Functions, luego en Multipliers y seleccionamos el core Multiplier.



Es interesante leerse y comprender la función que realiza y como la realiza. Podéis abrir los diferentes documentos y conocer en profundidad este Core. Haciendo doble clic, abriremos el asistente del Core seleccionado.



Mantenemos el nombre del core. Definimos la anchura de las señales A y B a 16 bits, y comprobamos en la pestaña IP symbol las entradas y salidas que tendrá nuestro core. Al decirle que utilice un multiplicador dedicado en lugar de uno basado en LUTs, en interesante observar como se utilizan exclusivamente XtremeDSP slices y ninguna LUT.



Dejando el resto de opciones por defecto, finalizamos el asistente y cerramos Core Generator con nuestro Core terminado.

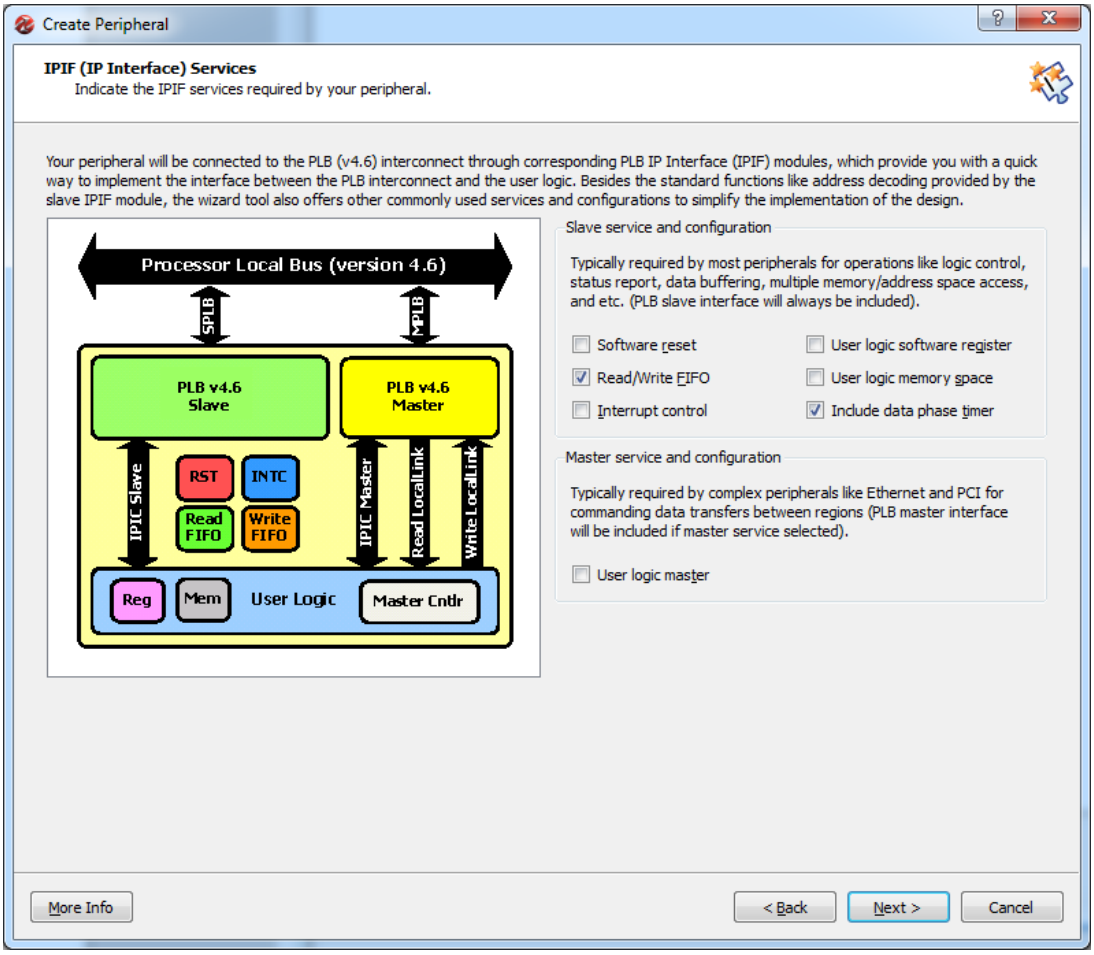
Para añadir dicho Core a MicroBlaze, seguiremos los siguientes pasos:

Lo primero será crear el hardware de MicroBlaze usando el puerto serie habitual, y sin necesidad de usar ningún periférico más. Creamos las aplicaciones de test y generamos nuestro sistema empotrado.

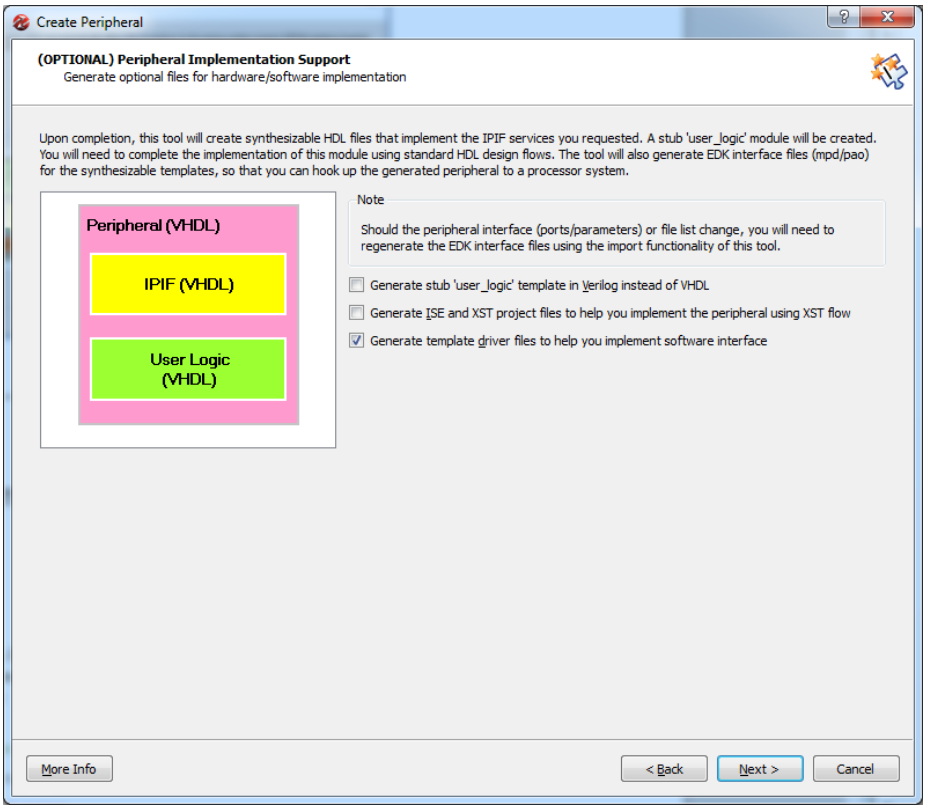
Una vez generado el bitstream y comprobado que no tenemos errores, en el menú Hardware escogemos Create or Import Peripheral.

Seleccionamos la creación de plantillas para un nuevo periférico y lo almacenamos dentro del directorio de nuestro Ejercicio_7. Le damos el nombre de mi_multiplicador y escogemos el bus PLB para su conexión con MicroBlaze.

Para el protocolo IPIF, seleccionamos la posibilidad de usar una FIFO para la lectura y escritura de los datos del Core del multiplicador. También seleccionamos la inclusión de un timer para su control (Include data phase timer) y deseleccionamos el uso del registro.



Dejamos el resto de opciones por defecto hasta la ventana de Peripheral Implementation Support, donde activamos la pestaña de Generate Template Drivers.



Ya tenemos creada la base del periférico, como queremos que dicho periférico funcione con los bloques FIFO creados, debemos modificar ciertas líneas del archivo user_logic.vhd. Abrimos dicho archivo y añadimos lo siguiente:

```
169
170   --USER signal declarations added here, as needed for user logic
171
172 component multiplier
173   port (
174     clk: IN std_logic;
175     a: IN std_logic_VECTOR(15 downto 0);
176     b: IN std_logic_VECTOR(15 downto 0);
177     p: OUT std_logic_VECTOR(31 downto 0));
178 end component;
179
180
```

```
191
192   --USER logic implementation added here
193
194 multiplier_0 : multiplier
195   port map (
196     clk => Bus2IP_Clk,
197     a => WFIFO2IP_Data(16 to 31),
198     b => WFIFO2IP_Data(0 to 15),
199     p => IP2RFIFO_Data);
200
```

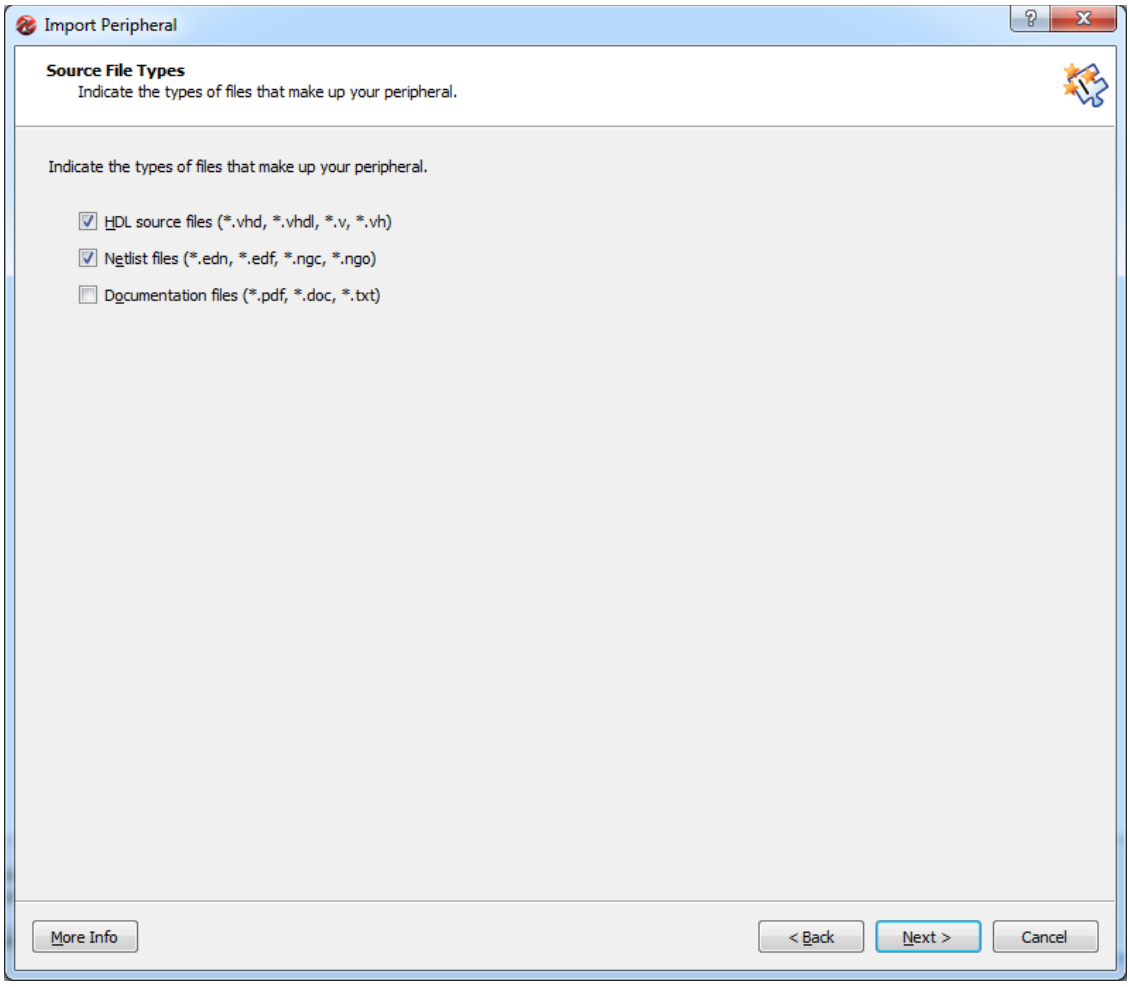
No debemos cambiar el nombre "multiplier" puesto que ese es el nombre que le da el Core Generator al multiplicador. También debemos comentar la siguiente línea:

```
272
273     --IP2RFIFO_Data <= WFIFO2IP_Data;
274
```

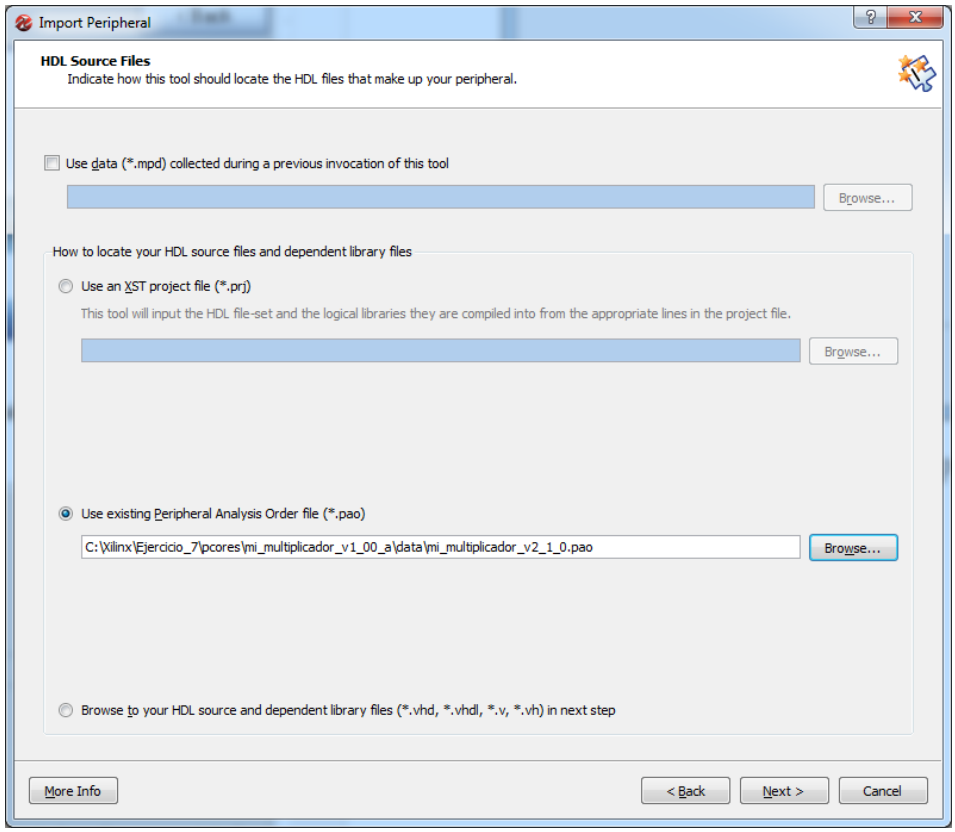
Con todo esto hemos conseguido que los valores que salgan de la FIFO pasen por nuestro multiplicador y que el resultado de dicha multiplicación se envíe de nuevo a la FIFO antes de enviarse al bus PLB.

Ahora deberemos incluir nuestro Core para que entienda estas líneas y funcione. Para ello, volvemos a abrir el Asistente para la Creación de Periféricos y seleccionamos la opción Import existing peripheral.

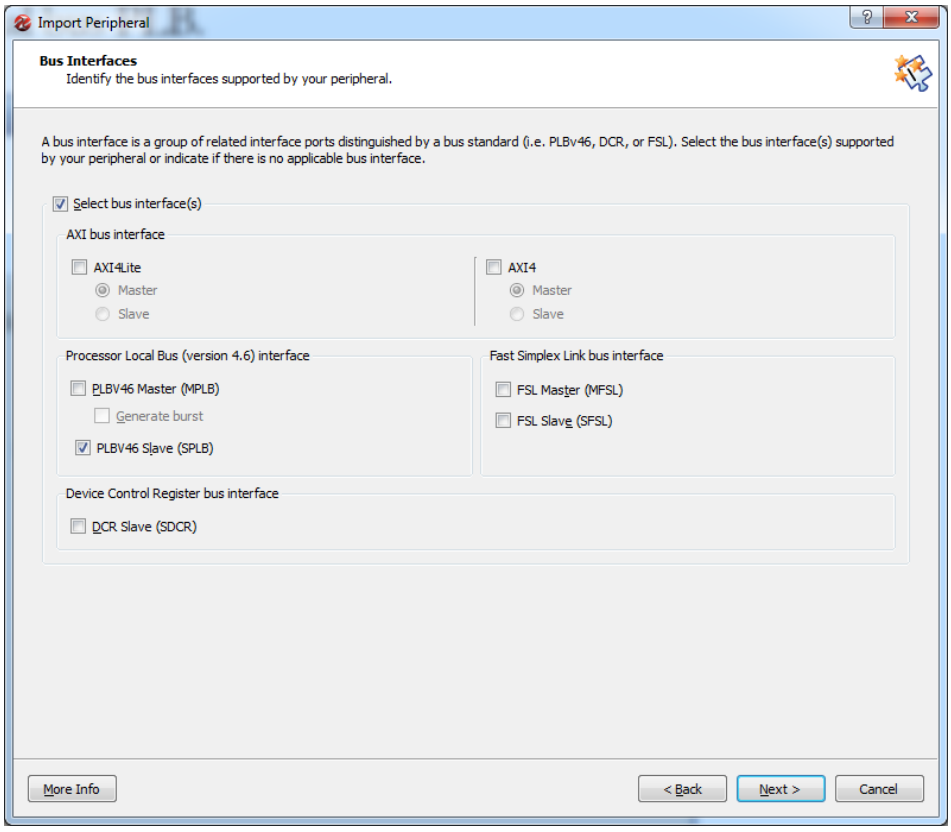
Importamos el periférico creado, con especial atención al número de versión, escogiendo según sea el periférico la opción de HDL o Netlist. Con la primera opción importaríamos periféricos creados en VHDL y con la segunda periféricos creados con Core Generator o con Simulink. En nuestro caso marcamos ambas opciones.



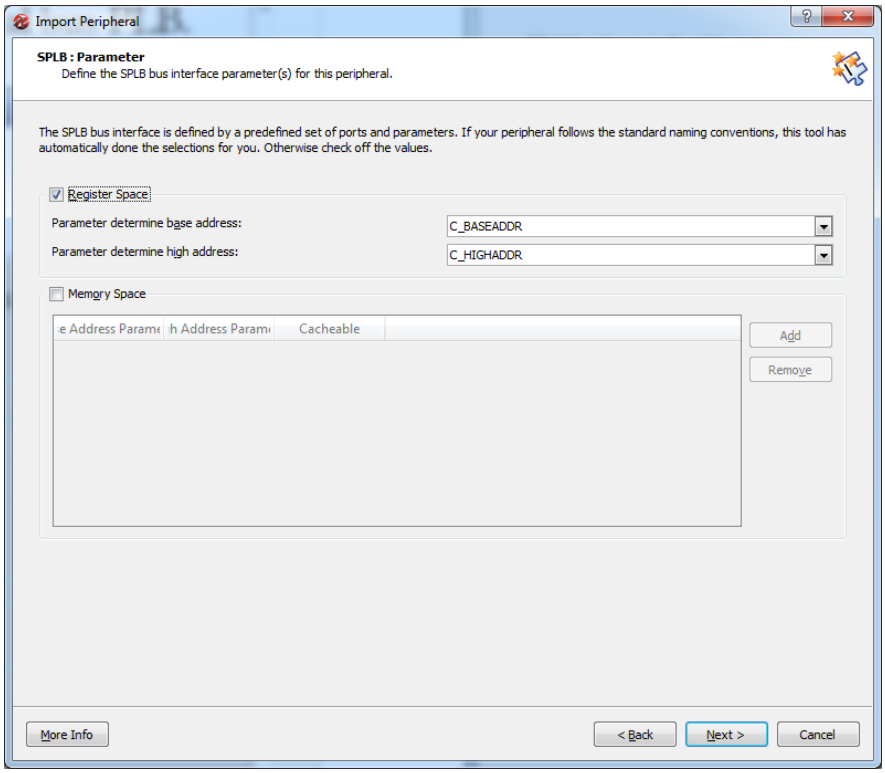
Seleccionamos el .pao de nuestro periférico para que entienda la estructura creada en el mismo.



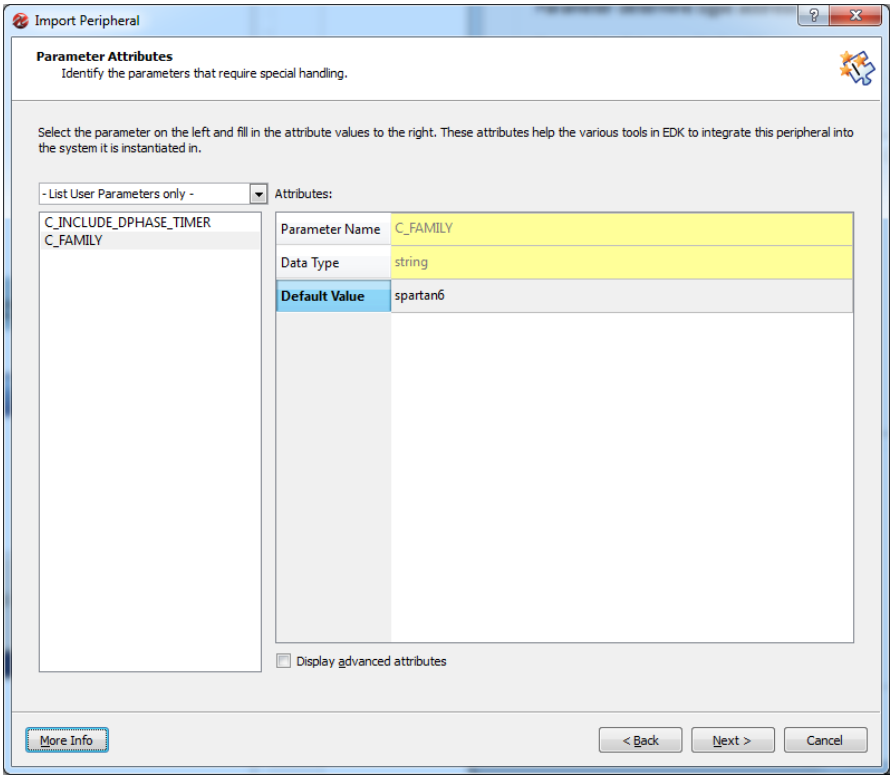
Seleccionamos la opción de conectar el periférico como esclavo al bus PLB.

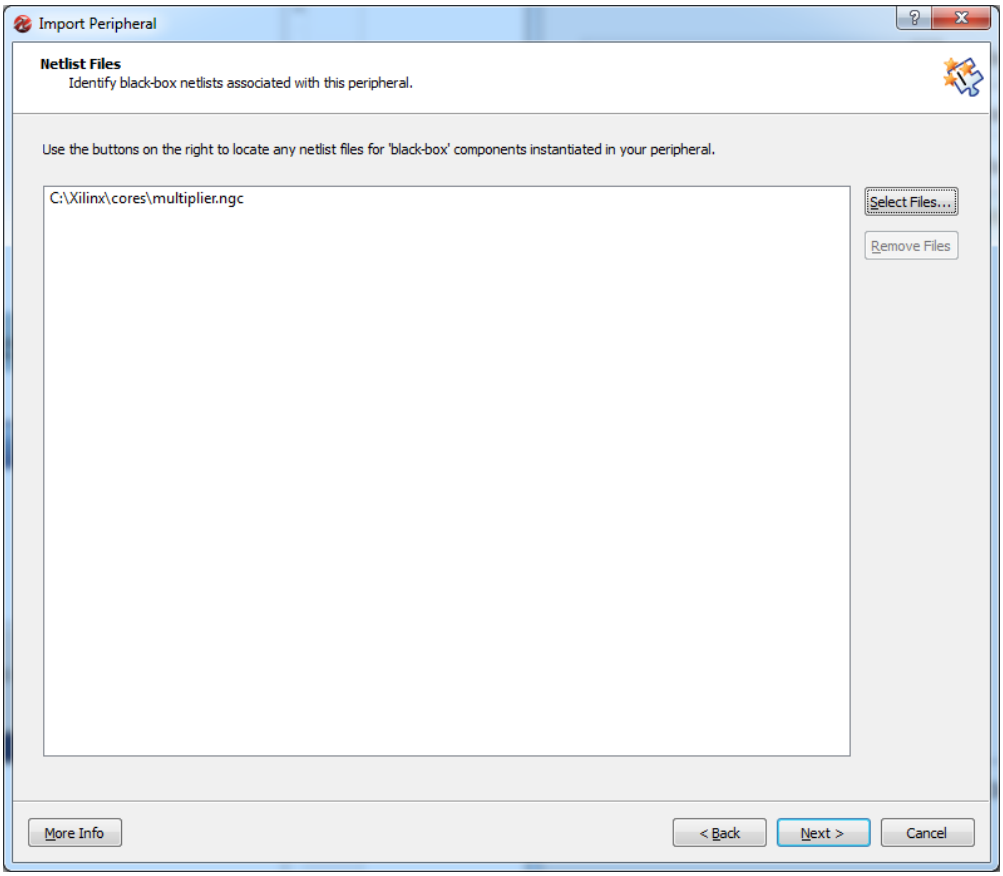


Definimos el espacio de registros:



Cambiamos la familia que viene por defecto a spartan6 y añadimos el .ngc creado con Core Generator que incluye nuestro periférico.





Finalizamos y ya tendremos un nuevo periférico creado a partir de un Core de ISE. Los mismos pasos se harían para incluir un periférico creado en VHDL.

Ya podemos añadir el bloque IP creado a nuestro sistema embebido. Conectándolo al bus, y dándole un tamaño de 64K dentro del mapa de memoria de MicroBlaze. Generamos el bitstream y comprobamos que no hay errores.

Sólo nos queda generar una aplicación software en C para comprobar su funcionamiento. Para ello, creamos en SDK una nueva aplicación software y añadimos los archivos mi_multiplicador.c y mi_multiplicador.h que se han creado automáticamente en la carpeta ..\Ejercicio_7\drivers\mi_multiplicador_v1_00_a\src

Reemplazamos el código que contiene por el siguiente:

```
#include "xparameters.h"
#include "xbasic_types.h"
#include "xstatus.h"
#include "mi_multiplicador.h"

Xuint32 *baseaddr_p = (Xuint32 *)XPAR_MULTIPLICADOR_0_BASEADDR;
```

```
int main (void) {

Xuint32 i;
Xuint32 temp;
Xuint32 baseaddr;

xil_printf("%c[2J",27);

// Comprobamos que el periférico existe

XASSERT_NONVOID(baseaddr_p != XNULL);
baseaddr = (Xuint32) baseaddr_p;
xil_printf("Test Multiplicador\n\r");

// Resetea la lectura y la escritura de las FIFOs
MI_MULTIPLICADOR_mResetWriteFIFO(baseaddr);
MI_MULTIPLICADOR_mResetReadFIFO(baseaddr);

// Escribe valores en la FIFO
for(i = 1; i <= 4; i++ ){
   temp = (i << 16) + i;
   xil_printf("Valores: 0x%08x \n\r", temp);
   MI_MULTIPLICADOR_mWriteToFIFO(baseaddr,0, temp);
}

// Lectura del valor de la FIFO
for(i = 0; i < 4; i++){
   temp = MI_MULTIPLICADOR_mReadFromFIFO(baseaddr,0);
   xil_printf("Multiplicacion: 0x%08x \n\r", temp);
}

// Resetea la lectura y la escritura de las FIFOs
MI_MULTIPLICADOR_mResetWriteFIFO(baseaddr);
MI_MULTIPLICADOR_mResetReadFIFO(baseaddr);
xil_printf("Fin Test\n\n\r");

// Permanece en un bucle infinito
while(1){
   }
}
```

Examina la salida del hyperterminal al ejecutar este código. ¿Entiendes lo que está pasando? ¿Cómo funciona la FIFO? ¿Por qué la última multiplicación no sale bien?

Se propone la generación de otro core, en este caso un comparador de 16 bits, y su introducción en el proyecto, de modo que compare los dos valores que se van a multiplicar y que indique en cada caso si el primer operando es mayor que el segundo o no.